



TECHNISCHE UNIVERSITÄT
CHEMNITZ

Fakultät für Informatik

CSR-25-02

Automotive Software Component for QT Based Car Status Visualization

Billava Vasantha Monisha · Hasan Saadi Jaber Aljaere ·
Wolfram Hardt

März 2025

Chemnitzer Informatik-Berichte



TECHNISCHE UNIVERSITÄT
CHEMNITZ

Automotive Software Component for QT Based Car Status Visualization

Master Thesis

Submitted in Fulfilment of the
Requirements for the Academic Degree
M.Sc.

Dept. of Computer Science
Chair of Computer Engineering

Submitted by: Billava Vasantha Monisha

Student ID: 758413

Date: 28.11.2024

Supervising tutor: Prof. Dr. Dr. h. c. W. Hardt

Hasan Saadi Jaber Aljzaere, Master

Acknowledgements

I begin by offering my profound thanks to Almighty, who cleared the way and gave me the energy and endurance to complete this thesis. I also want to extend my heartfelt thanks to my parents, whose continuous support, encouragement, and guidance enabled me to pursue this journey.

Especially, I would like to express my deepest gratitude to Prof. Dr. Dr. h. c. Wolfram Hardt, under whose supervision I had the great opportunity to carry out this work. His precious hints, patience, and knowledge have always been very important to find the way through the different sections of this thesis.

A special word of thanks to my supervisor: Hasan Aljaere, whose continued support, feedback, and guidance had great importance for this work. His technical insights and constructive suggestions were quite valuable with respect to the direction of my research. I would also like to thank Murat Sevil for the precious documentation support and pioneering work he has done on AUTOSAR, which served as the basis for this master thesis. His contribution significantly set the frame of this work. I also want to thank the staff of the Department of Computer Engineering, Technische Universität Chemnitz for not only sustained support, but especially for creating an enabling environment for research and learning.

Lastly, I am very thankful to all my friends and colleagues who inspired me, supported me technically, and encouraged me morally throughout the journey. Their presence and encouragement provided a source of strength, particularly during the critical phases of this work. I am eternally grateful for those contributions and support, and this thesis would not be possible without the ones above.

Abstract

Nowadays, automotive systems are getting much more complicated, so they require solid bases for the communication of current data in real time, their processing, and visualization. The subject of this thesis is the integration of AUTomotive Open System Architecture (AUTOSAR) software components with the QT-based application for real-time vehicle status visualization. Controller Area Network (CAN) bus communication is implemented within the TUCminiCar system, along with a setup using Raspberry Pi, which would be used for forwarding and displaying key metrics of speed, steering angle, and proximity data.

It considers three major components: an Aurix microcontroller running AUTOSAR Software components with sensor data processing, a Raspberry Pi interfaced with the CAN bus using PiCAN2 hardware and forwarding via Wi-Fi, and a CEBox hosting a QT application for real-time display.

In addition, this thesis further develops an easier-to-use dashboard on QT 5.15 for the representation of essential vehicle data in real time. The system shall guarantee low-latency communication between the Raspberry Pi and the CE Box to ensure smooth visualized data updates.

This thesis will use existing AUTOSAR SWCs in the TUCminiCar platform and implement real-time data acquisition, CAN message processing, and error handling of a QT-based visualization dashboard. In this project, it was shown how integration of an AUTOSAR-compliant setup with a custom QT interface allows for effective real-time monitoring of vehicle metrics. Performance analysis and functional testing validate the system's responsiveness, pointing to scalability in handling data for more complex automotive applications, with respect to security and visualization options.

Keywords: AUTOSAR, QT framework, CAN bus, Car Status Visualization, CE-Box.

Zusammenfassung

Heutzutage werden Automobilsysteme immer komplexer und erfordern daher solide Grundlagen für die Kommunikation aktueller Daten in Echtzeit, deren Verarbeitung und Visualisierung. Das besondere Thema dieser Arbeit ist die Integration von AUTOSAR-Softwarekomponenten mit der QT-basierten Anwendung zur Echtzeitvisualisierung des Fahrzeugstatus. Die CAN-Bus-Kommunikation ist im TUCminiCar System implementiert, zusammen mit einem Setup mit Raspberry Pi, das für die Weiterleitung und Anzeige wichtiger Geschwindigkeits-, Lenkwinkel- und Annäherungsdaten verwendet wird.

Es berücksichtigt drei Hauptkomponenten: einen Aurix-Mikrocontroller, auf dem AUTOSAR-SWCs mit Sensordatenverarbeitung laufen, ein Raspberry Pi, der über PiCAN2-Hardware und Weiterleitung über WLAN mit dem CAN-Bus verbunden ist, und eine CEBox, die eine QT-Anwendung für die Echtzeitanzeige hostet.

Darüber hinaus wird in dieser Arbeit ein benutzerfreundlicheres Dashboard auf QT 5.15 zur Darstellung wesentlicher Fahrzeugdaten in Echtzeit weiterentwickelt. Das System soll eine Kommunikation mit geringer Latenz zwischen dem Raspberry Pi und der CE-Box gewährleisten, um eine reibungslose Aktualisierung der visualisierten Daten zu gewährleisten.

Diese Arbeit wird bestehende AUTOSAR-SWCs in der TUCminiCar Plattform nutzen und Echtzeit-Datenerfassung, CAN-Nachrichtenverarbeitung und Fehlerbehandlung eines QT-basierten Visualisierungs-Dashboards implementieren. In diesem Projekt wurde gezeigt, wie die Integration eines AUTOSAR-kompatiblen Setups mit einer benutzerdefinierten QT-Schnittstelle eine effektive Echtzeitüberwachung von Fahrzeugmetriken ermöglicht. Leistungsanalysen und Funktionstests validieren die Reaktionsfähigkeit des Systems und weisen auf Skalierbarkeit bei der Datenverarbeitung für komplexere Automobilanwendungen im Hinblick auf Sicherheits- und Visualisierungsoptionen hin.

Schlüsselwörter: AUTOSAR, QT-Framework, CAN-Bus, Car Status isualization, CE-Box.

Content

Acknowledgements	1
Abstract	2
Zusammenfassung	3
Content	4
List of Figures	7
List of Tables	8
List of Abbreviations	9
1 Introduction	10
1.1 Background	11
1.1.1 Automotive Systems	11
1.1.2 AUTOSAR- Automotive Open System Architecture	13
1.1.3 CAN Bus (Controller Area Network)	15
1.1.4 QT Framework	16
1.2 Problem Statements	21
1.3 Motivation	22
2 State of Art	24
2.1 Evolution of Automotive User Interface	24
2.1.1 Early Automotive User Interfaces	24
2.1.2 Rise of Digital Displays and Infotainment	24
2.2 Current Industry Tools for Automotive HMI Design	25
2.3 CAN Bus in Infotainment Systems	27
2.4 Overview of Qt in Embedded Systems Development	28
2.4.1 Qt for Automotive UI`s	29
2.4.2 Qt in Embedded Linux Systems	30
2.5 AUTOSAR and QT based Visualization	31
3 System Architecture	33
3.1 Overview of the TUCminiCar	33

3.2	Hardware Components	34
3.2.1	Aurix Microcontroller	34
3.2.2	Raspberry Pi and Pican2	35
3.2.3	Sensors Used in the TUC Mini Car	37
3.2.4	CE-Box	38
3.3	Software Components	39
3.3.1	Autosar Software Components of TUCMiniCar	39
3.3.2	CAN Communication and Message Flow	41
3.3.3	Data Flow and Integration with QT Application	44
4	Design and Implementation	47
4.1	System Requirements	47
4.1.1	Hardware Requirements	47
4.1.2	Software Requirements	48
4.2	System Design Overview	48
4.3	Gateway Application on Raspberry Pi	51
4.3.1	Overview of the Gateway Application	52
4.3.2	Gateway Application Structure and Code	52
4.4	Visualization Application on the CE-Box	54
4.4.1	Queued Connection and Single-Slot Mechanism	57
4.4.2	Class Diagram for the Dashboard application	58
4.5	Network Configuration and Error Handling	59
4.6	Sequence Diagram	61
4.7	Dashboard UI Design and Layout	63
5	Testing and Evaluation	66
5.1	Testing Environment Setup	66
5.1.1	Hardware setup	66
5.1.2	Data Simulation Application	67
5.2	Functional Testing of the Qt Application	69
5.2.1	Basic Functional Tests	70
5.2.2	Error and Alert Display Tests	75

5.3	Testing with Tiny-Can.....	76
5.4	Performance Testing.....	78
5.5	Observations from testing on the TUCminiCar.....	80
5.6	Limitation.....	81
6	Conclusion and Future Work.....	83
6.1	Conclusion	83
6.2	Future Work	83
7	Bibliography	85
	Appendix A – Qt framework.....	89
	Appendix B- Application Configuration	92
	Appendix C – Code Snippets	95
	Appendix C – QR Codes	97

List of Figures

Figure 1.1: Embedded Systems used in Automotive. [6]	12
Figure 1.2: AUTOSAR Architecture. [5].....	14
Figure 1.3: Qt Framework. [15].....	18
Figure 1.4: QT creator GUI.....	20
Figure 2.1: Interior of Mercedes SL from 1970s compared to current model. [20].....	25
Figure 2.2: Instrumental Cluster designed by Kanzi. [32]	27
Figure 2.3: CGI based Instrumental cluster. [31]	27
Figure 2.4: GUI that appears on the Dashboard. [34].....	28
Figure 2.5: Black Pearl Display Home. [35]	29
Figure 2.6: View of the Demonstrator. [36].....	30
Figure 2.7: Simulation Platform. [40]	32
Figure 3.1: Overview of the TUCminiCar.....	34
Figure 3.2: Car Demonstrator- TUCminiCar.....	34
Figure 3.3:AURIX™ TC387. [41]	35
Figure 3.4: Raspberry Pi 3b+ and PiCAN2. [42] [43]	36
Figure 3.5: Ultrasonic Sensor (HRLV-ShortRange-EZ). [44]	37
Figure 3.6: LiDAR Sensor (LD19). [45].....	38
Figure 3.7: The CE-Box and the front panel connection. [46].....	39
Figure 3.8: Overview of the available software components.	41
Figure 3.9: CAN Message table for CS1 from D0-D4.....	43
Figure 3.10: CAN message table for CS1 from D5-D7.....	43
Figure 4.1: Class Diagram of Dashboard Application.....	59
Figure 4.2: Sequence diagram for the application.	61
Figure 5.1: Setup for testing.	67
Figure 5.2: UI of the Data Simulator.	69
Figure 5.3: All elements visible and functional.....	70
Figure 5.4: Distance sensor data when speed <60.	72
Figure 5.5: Lidar data displayed when speed >60.....	72
Figure 5.6: Battery percentage, break lights and engine power.	73
Figure 5.7: Steering and the lights.....	74
Figure 5.8: Sensors unavailability.....	75
Figure 5.9: When battery 0%.....	76
Figure 5.10: TUCminiCar with the Tiny Can hardware connected.....	76
Figure 5.11: Tiny CAN GUI.....	77

List of Tables

Table 2.1: Overview of Commercial Software Frameworks for Automotive HMI Development	26
Table 4.1: Hardware Requirements.....	47
Table 4.2: Hardware Requirements.....	48
Table 4.3:Requirements of the Qt Visualization.	49
Table 4.4: Requirements for the Gateway application.....	52
Table 4.5: UI Components Overview.....	63

List of Abbreviations

ADAS	Advanced Driver Assistance Systems
AUTOSAR	Automotive Open System Architecture
BSW	Basic Software
CAN	Controller Area Network
DTC	Diagnostic Trouble Code
ECU	Electronic Control Unit
GUI	Graphical User Interface
HMI	Human-Machine Interface
HUD	Head-Up Display
IDE	Integrated Development Environment
IOT	Internet of Things
LIDAR	Light Detection and Ranging
MVC	Model-View-Controller
OS	Operating System
RAM	Random Access Memory
RPI	Raspberry Pi
RTE	Runtime Environment
SOM	System on Module
SWC	Software Component
TUC	Technische Universität Chemnitz
UDP	User Datagram Protocol
UI	User Interface
VFB	Virtual Functional Bus
XML	Extensible Markup Language

1 Introduction

The automotive industry is fast changing into a software-driven field, and huge growth is estimated in the automotive software market. The global size of the automotive software market is projected to reach approximately USD \$48.7 billion by 2030, growing at a compound annual growth rate (CAGR) of 7.6% from 2023 onwards [1]. The complexity in such systems calls for steadfast software frameworks that can provide assurance of the correct, real-time exchange of data among the various vehicle systems. AUTOSAR is becoming the standard for a fully modular software architecture to improve compatibility and decrease integration complexities between Electronic Control Units, ECUs [2].

Another key element is real-time visualization, with electric and autonomous vehicles, where instant feedback on battery health, efficiency, and safety diagnostics are critical. An estimation of 2030 puts almost all new vehicles on the road with some form of real-time data monitoring interface aimed at raising driver awareness and support for autonomous functionalities. Frameworks like those mentioned create a foundation for developing these interfaces, enabling intuitive, digital dashboards that present essential information to contribute to safer, better-informed driving experiences.

However, all these technologies are usually very costly, so small research institutions and developers cannot afford them. Therefore, this research tries to fill this gap with the design of a low-cost, AUTOSAR-compliant, real-time visualization system based on the QT framework specifically for the TUCminiCar, which is an educational and research-oriented platform. This project presents a low-cost, modular solution using the CAN bus protocol and a Raspberry Pi that satisfies industry standards for real-time automotive data monitoring.

We want to demonstrate with this work that open-source, accessible platforms can deliver real-time vehicle data visualization equal to proprietary systems. This project not only furthers the state of the art in automotive software research but also pushes toward the broader accessibility of educational and experimental applications. The QT framework will be used in developing an AUTOSAR-based real-time car status visualization system for the TUCminiCar in this thesis. After this, the motivation, background, problem statement, and layout of the thesis will follow.

1.1 Background

The background section sets the technical basis that can assist in understanding the key elements supporting the research project. The thesis targets the integration of AUTOSAR software components with the QT framework for real-time visualization of vehicle status. Subsequent subsections develop key technologies that make this integration feasible.

1.1.1 Automotive Systems

In the last couple of decades, modern vehicles have gone through considerable changes in the automotive industry, from complex embedded systems capable of coping with a wide variety of applications, starting from engine management up to advanced driver assistance. In an automotive context, a system using hardware and software integrated to perform specific tasks within a vehicle can be identified as an embedded system based on a microprocessor-based control system. These systems control a wide range of functions, from basic vehicle controls like engine and brakes to advanced systems like infotainment and ADAS [3]. Along with time the complexity of embedded systems has also increased.

Key current automotive development trends shaped a shift towards software-defined vehicles. That would mean most of the vehicle functions will now be controlled by software rather than hardware. It is described that the shift of automotive systems towards a software-centric approach has resulted in the demand for standardized frameworks like AUTOSAR to manage the growing complexity of electronics inside the vehicle [4]. This, in turn, has created further innovation in this area, as an increasing number of real-time visualization data systems is required, allowing drivers and technicians to have an intuitive interface showing the insight of key vehicle parameters. The key functions of Automotive Embedded Systems are discussed below:

Real-time Data Processing: Automotive embedded systems operate and process real-time data. In fact, braking, steering, and collision detection are all such applications that rely on the response time of the system; even a fraction of delay may result in an accident. Such systems should respond to sensor inputs in milliseconds to ensure vehicle performance and safety.

Reliability and Safety in System: Most functions in a vehicle are critical; hence, automotive embedded systems must be highly reliable. Systems responsible for airbags, braking, and engine management just cannot afford to fail because the

consequences might be disastrous. The software of such systems is put through rigorous testing and validation with strict testing about safety.

Scalability and Modularity: These systems must be inherently developed in vehicles with the increase in the features that they possess. To achieve this, frameworks like AUTOSAR provided a standard way of developing and integrating software. The AUTOSAR helps manufacturers build modular systems in which individual software components can be shared across different models, thereby reducing the cost and complexity.

Development of Automotive Embedded Software:

Automotive embedded software development is complex and specialized, relying heavily on software engineering and automotive technologies. Embedded software plays the most important role in-vehicle, managing everything from the more basic functions of engine control and climate management to the more advanced capabilities of autonomous driving and vehicle connectivity.

This covers the design, coding, testing, and maintenance of software to run embedded systems. Automotive embedded software, especially for life-critical systems such as airbags and braking systems, is set at a very high bar for safety and reliability. Moreover, with consumers refusing to yield in their demands for more features-smartphone integration, real-time traffic updates, voice recognition-the automotive-inspired software development complexity grows day in and day out.

AUTOSAR has considerably simplified embedded software development in the automotive domain. AUTOSAR provides the right platform for developing modular and scalable software that can be reused across models and brands [5]. It saves time and cost from developing new software every time from scratch and enhances the reliability and compatibility of the software.



Figure 1.1: Embedded Systems used in Automotive. [6]

1.1.2 AUTOSAR- Automotive Open System Architecture

As discussed in the previous section about AUTOSAR, it has an architecture split into several layers, allowing for the abstraction of hardware and the consequent separation of software components. These layers work together to provide the flexibility and modularity needed to support the development of complex vehicle systems. Key layers include: [5]

The Application Layer represents the topmost layer, which holds the actual functional software components that execute a variety of in-vehicle tasks, ranging from engine control and braking systems to infotainment and ADAS. Each SWC in this layer is, by definition, hardware-independent and thus shareable across multiple vehicle models. This modularity reduces not only development costs but also the need for major rework in case of any adaptation or upgrade of the software components.

Beneath the Application Layer, the Runtime Environment provides the communication interface for both the Application Layer to all the software underneath and vice-versa. The RTE realizes transparency in data exchange amongst the SWCs and, extending further downwards, transparency right down to the Basic Software. This abstraction will provide better flexibility and scalability while communications between software components will happen, not having to deal explicitly with some hardware-specific features of the hardware. This is enabled by standardized communication through the Virtual Functional Bus, a decoupling layer between the software and hardware.

The Basic Software layer offers fundamental services, which are the mainstay required for the execution of the Application Layer. This includes services related to memory management, handling of input and output, communication protocols such as CAN, LIN, and Ethernet. Indeed, at the BSW, the system can ensure it will reliably interface with sensors, actuators, and other external devices in a harmonious way. Additionally, included is diagnostics and error handling to further improve overall system reliability and safety.

AUTOSAR MCAL is the core-layer entity in the AUTOSAR architecture, decoupling the hardware-specific information of higher layers of the software stack. It represents the standard interface to access the peripheral on the microcontroller-such as timers, ADC, and communication interfaces-for operating systems and higher application layer software. This abstractive functionality from MCAL therefore means that for any software developer, it will be much easier to port software into other microcontroller platforms, an activity that will entail low effort. But this abstraction of hardware is not

to be overlooked as such since scalability and reusability are indeed the hallmarks of the AUTOSAR framework.

With integration, these layers ensure highly modular, scalable, and reliable systems based on AUTOSAR. This architecture therefore provides another standardized approach whereby automotive software development enables a number of vendors to potentially collaborate without the complexities seen in the integration of added functionalities. Moreover, this decouples software from hardware through AUTOSAR, leading to enhanced automotive features whereby compatibility will remain with existing systems. Most importantly, however, AUTOSAR provided an excellent starting point, mainly in its layered architecture for the doing of modern automotive software: in other words, module design, hardware abstraction, and emphasizing standardization-are enablers to be enabled in today's ever complex and software driven vehicle.

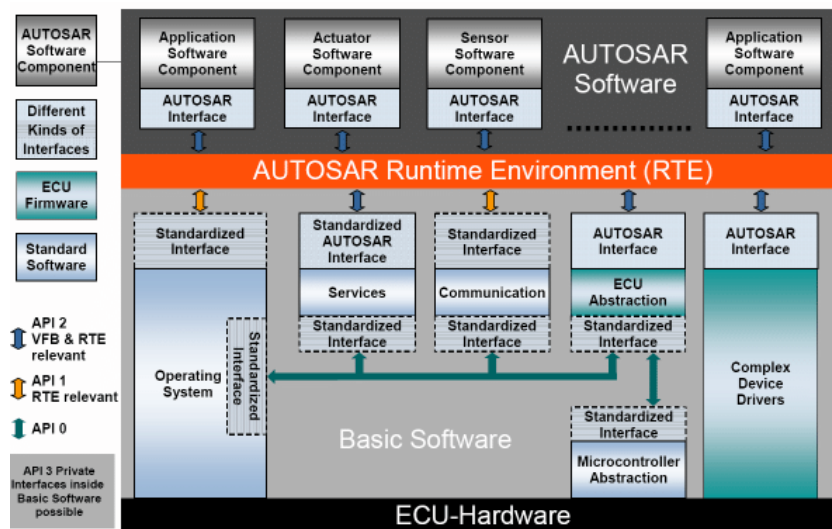


Figure 1.2: AUTOSAR Architecture. [5]

AUTOSAR Software Components (SWCs)

AUTOSAR Software Components (SWCs) are key elements of the AUTOSAR architecture, designed to perform specific tasks within a vehicle's control system. These components are reusable, scalable, and hardware-independent, allowing them to be easily integrated into different vehicle platforms with minimal modifications. Each SWC communicates with other components and hardware through the Runtime Environment (RTE), which manages data exchange and communication between SWCs and the hardware. Standardized interfaces in the RTE ensure SWCs are independent of the microcontroller or ECU hardware specifics, making them modular and portable. Key Features of AUTOSAR SWCs includes:

Reusability: SWCs can be developed once and reused across different vehicle platforms, reducing development time and costs, and simplifying maintenance and upgrades.

Interoperability: AUTOSAR SWCs use standardized interfaces, allowing seamless communication between components and ECUs, regardless of the hardware or software supplier.

Scalability: SWCs handle a wide range of functionalities, from simple tasks like controlling lights and wipers to complex functions like adaptive cruise control and collision avoidance.

Decoupling of Hardware and Software: By abstracting the hardware layer, SWCs enable independent software development, improving collaboration between OEMs and suppliers. [7]

Below are the few examples of the SWC Applications:

Engine Management: SWCs optimize engine fuel injection, combustion, and emissions in real time, ensuring efficient performance under various driving conditions.

Brake Control Systems: In anti-lock braking systems (ABS), SWCs monitor wheel speed and apply brake pressure to prevent skidding.

Infotainment Systems: SWCs manage multimedia and connectivity features, integrating navigation, audio control, and Bluetooth connectivity into the central display interface in modern vehicles. [8]

1.1.3 CAN Bus (Controller Area Network)

The Controller Area Network bus represents the most used protocol in automotive systems to efficiently manage real-time inter-communication among multiple vehicle subsystems. This kind of protocol allows various Electronic Control Units operating over a centralized network to interchange data with parameters like vehicle speed, steering angle, and engine performance. Besides, CAN bus is apt for tackling the huge communicational difficulties undertaken by modern vehicle types with a lot of electronic control units onboard due to its high degree of error-checking and scalability [9] Also, low-cost CAN bus systems are able to implement a CAN bus system using Raspberry Pi for monitoring vehicle parameters using embedded systems. Research has shown this to be possible. [10]

Having started in 1986 from Bosch, nowadays, CAN has grown to become an industrial standard for in-vehicle communications, especially in automotive and industrial applications. In-vehicle ECUs have the capability to broadcast information

in real time and with high integrity of the data while reducing wiring complexity for a CAN bus connected to units such as the engine, transmission, brakes, and infotainment systems. Rajasekar & Bhaskar [11] gave evidence that the merits of CAN bus are high fault tolerance and real-time processing capability; thus, CAN bus is very suitable for systems that need safety and performance. Presently, the vehicle uses CAN to communicate with safety-critical systems like ABS, airbag control, and engine management but is also applied to less critical systems, including infotainment and navigation. The multi-master topology was designed to allow ECUs in CAN to perform communication with no central arbitrator for better flexibility in a system.

Challenges and Future Trends in CAN Bus

Though CAN protocol remains a widely utilized communication standard in automotive industries, challenges do occur. One problem with CAN is its bandwidth limitation and, therefore, can be a bottleneck in the automotive environment where high bandwidth data communication becomes essential, for instance, in autonomous driving or advanced infotainment. Hence, the more and more vehicles started connecting to everything and feature-rich, higher data transfer rates started motor running more advanced protocol developments such as CAN FD. CAN FD can be thought of as an extension to CAN in the sense that it resolves bandwidth limitations of classic CAN by simply increasing the size of the data payload. Another emerging challenge is the inclusion of security features within CAN systems. While modern vehicles are connected to external networks, they will be more exposed to cyber-attacks [12]. Current efforts within the automotive industry are toward improvement in security features for the CAN communication system to avoid unauthorized access or manipulation of critical information about the vehicle. [13]

Despite that, the basic features of CAN addressing real-time fault-tolerant and low-cost issues are reasons why, up to this date, it is paramount in automotive use. Further steps in the development of CAN technology will continue into the increasing demand for electric vehicles, autonomous cars, and connected vehicles to ensure communication efficiently and reliably.

1.1.4 QT Framework

Qt framework [14] is a powerful toolkit intended for cross-platform, rich GUI-based application development; it perfectly fits with automotive applications because of its modularity, high-performance capability, and robust visualization option. Qt's flexibility allows developers to create customized interfaces that present complex

vehicle data, such as speed, engine status, and sensor information in a visually appealing and responsive way. Its ability to efficiently manage real-time data has made Qt increasingly popular in the automotive sector.

Qt is known for its cross-platform development features, which makes sure that applications will work seamlessly across multiple operating systems, from Windows, macOS, and Linux to iOS and Android. Its philosophy of "write once, deploy anywhere" supports efficient development by allowing a single code base with minimal modifications to be maintained across different platforms. Qt is coded mainly in C++, but it also extends the language with features such as "Signals and Slots" that make event handling easier to work with. Compatibility with big compilers like GCC, Clang, MinGW, and MSVC extends its use for many more environments [14].

Qt contains its own build tool, namely `qmake`, to simplify development by making cross-platform builds easier, acting mostly as a frontend for native build systems such as CMake, Make, and MSVS. Also, Qt has an Integrated Development Environment, **Qt Creator**, introduced in 2009 [15]. Though this IDE is targeted at Qt development, the framework itself can easily be used with other popular IDEs such as Microsoft Visual Studio; thus, it leaves all choices open for the developer.

But Qt's huge ecosystem of libraries doesn't stop at GUI development: database management, networking, graphics and commercial ones, too, so a developer can choose if he wants to use one license or another depending on his project. This flexibility in licensing, combined with its rich features, makes Qt very attractive for embedded automotive applications where adaptive and robust interfaces are needed.

This thesis covers the use of Qt for the development of a real-time visualization interface that is to be supported for data from the CAN bus, processed within an AUTOSAR system, and displayed on the Raspberry Pi. With the vast variety of libraries and modules provided in Qt, support could be provided for the introduction of visual elements like speedometers, gauges, and indicator lights, hence offering a user-friendly interface while monitoring vehicle status.

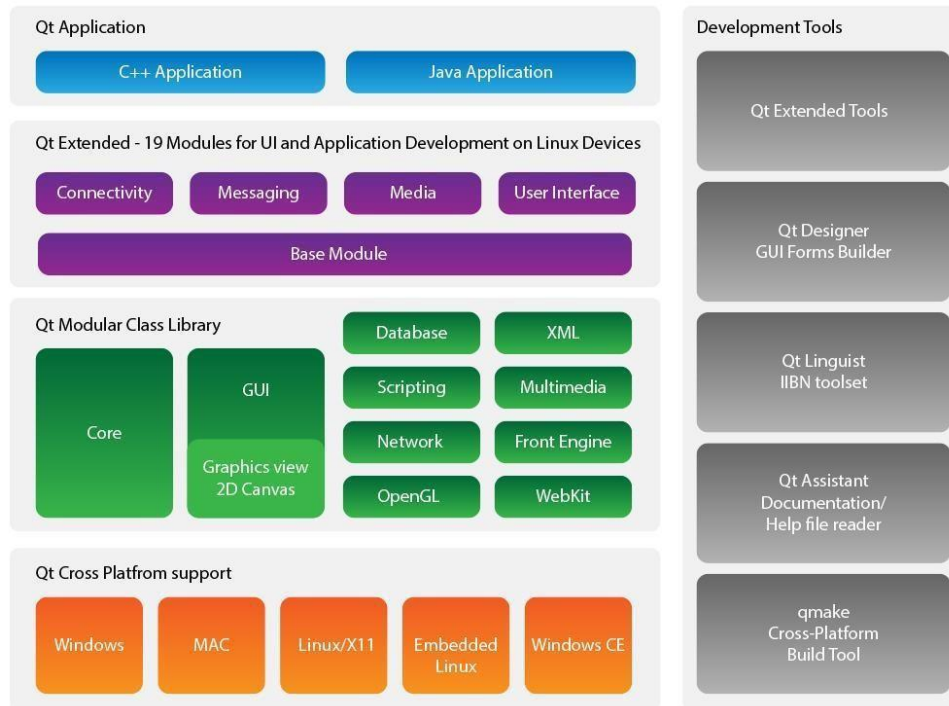


Figure 1.3: Qt Framework. [15]

QML

In 2009, Qt added a declarative software framework called Qt Quick and its own markup language, QML [14], so that Qt would be more effective in the creation of modern, touch-based, and interactive GUIs. QML was mainly used for the definition and structuring of graphical user interfaces, including controls, signals, and visual effects. It has been designed to handle touch input along with fluid graphics rendering, and thus enables dynamic visualizations in 2D/3D along with animations.

Qt Quick works in tandem with JavaScript on its V4 JavaScript engine, allowing the developer to put logic within the QML files themselves for the most agile and responsive UI designs. This capability of JavaScript within QML also facilitates easy integration with the back end since QML can easily interface with C++ classes and functions, making it a very powerful link between the UI and the underlying application logic. Furthermore, QML organizes interface elements as a tree for interfaces, allowing modular construction of intricate UIs in a very natural way.

The example below will have a TUC logo in the background with a clickable button is an simple example of QML application with Qt framework.

```

import QtQuick 2.15
Rectangle {
    width: 300
    height: 250
    color: "white"
    Image {
        source: "img/TUClogo.png"
        anchors.fill: parent
        opacity: 0.2 // Background logo with reduced opacity
    }
    Text {
        text: "Welcome"
        anchors.centerIn: parent
        font.pixelSize: 20
        color: "black"
        MouseArea {
            anchors.fill: parent
            onClicked: {
                console.log("Welcome to TU Chemnitz!")
            }
        }
    }
}

```

QML Elements

QML elements can be classified as either visual elements or non-visual elements [14]: Visual elements - such as Rectangle - have graphical properties, and typically render a graphical area on screen. Non-visual elements - such as Timer - provide underpinning functionality which can be used to control or to manipulate the properties of other visual elements. QML enables developers to define both the layout and behaviour of an interface, exposing them to dynamic, interactive user interfaces through the combination of visual and non-visual elements. QML elements are connected through unique identifiers. Thus, it becomes easy to access the properties of one element from another and manipulate them.

QML Components

A QML component [14] is any reusable chunk of QML that can be declared as a single, self-contained file. To create a file-based component, the QML element itself must be placed into a single file with the given filename, thus enabling it to be reused in other QML files. This kind of component-based architecture Favours modularity and, consequently, allows dealing with and reusing complex UI elements effectively when embedding them into a larger application. Once defined, a QML component may be instantiated like any other QML element. This makes it easier to structure and scale during the design of the UI.

Qt Creator

Qt Creator is a cross-platform IDE [16] self-contained for developing computer applications. Qt Creator provides the leading functionalities required to enhance the productivity of developers working with this IDE: a code editor featuring C++, QML, JavaScript, and ECMAScript support, rapid code navigation tools, source code refactoring, and integrated debugging. It also provides a powerful GUI designer and code analyzer, which make the development of visually rich, responsive applications much easier. Qt Creator also supports remote deployment-target device assembly and the deployment directly within the development environment to enable smooth integration and testing.

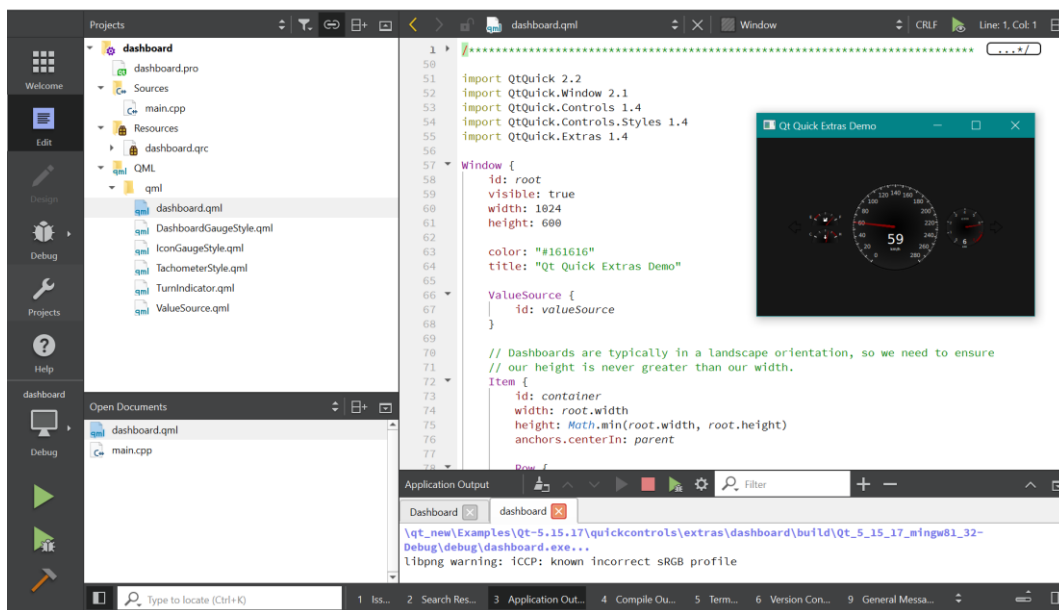


Figure 1.4: QT creator GUI.

Why QT 5.15?

Qt 5.15 [17] is very suitable for development on Raspberry Pi 3B+ since this edition is close to the hardware capabilities of Raspberry Pi and enjoys broad community support; this version is under long-term support. This would make setting up cross-compilation environments easier and would make deployment of applications to embedded Linux systems like Raspberry Pi more effective.

Some of the major features of Qt 5.15, which make it even more suitable for development on Raspberry Pi, are as follows: Improved GPU and OpenGL ES Support Qt 5.15 brings in improvements in hardware-accelerated graphics, very important for responsive applications on embedded devices. It includes support for the Mesa V3D graphics stack when working with Raspberry Pi, hence enabling

efficient OpenGL rendering - an essential requirement for applications that make use of smooth real-time graphics performance.

Qt 5.15, on its own, has toolchains and configurations to make deployment seamless on ARM-based architectures like the Raspberry Pi. Moreover, these configurations can be fine-tuned, and a developer can only include modules that will be required by an application. A development result of this nature could lead to quite light applications with a smaller footprint, ideal for resource-constrained systems like those found in embedded systems.

1.2 Problem Statements

The application of QT-based visualization in the AUTOSAR framework poses enormous technical challenges due to the intrinsic complexity of merging real-time data management with intuitive visual interfaces. AUTOSAR has been found in wide application within automotive systems because it has a standardized approach toward software development, offering a modular and scalable platform for the communication of ECUs in managing vehicle operations. While excellent in AUTOSAR for data transmission and inter-subsystem communication, it lacks native support for visualization frameworks like QT, which is crucial for modern car status monitoring.

What is more, within the TUCminiCar a research demonstrator from automotive environments-there is still no AUTOSAR software component that would realize real-time visualization of car status. That is, there is a lack of a mechanism to present, via a graphical interface, the input from sensors available for speed, steering, lights, battery level, power, and engine status. This denies the possibility of real-time monitoring of the performance of the vehicle itself, which becomes necessary in research or educational applications when intuitive data representation plays such an important role regarding analysis and diagnostics. This represents the gap in usability from the existing platform and opens a more communicative instructive environment for research. Data visualization in real time facilitates better learning experiences.

This thesis will attempt to bridge the gap through the design of a QT-based application integrated with AUTOSAR environment to realize real-time visualization of vehicle data. In consideration of the limited operating capability of the TUCminiCar on Aurix and Raspberry Pi 3B+, transmitting data using Wi-Fi, it would be challenging how big the AUTOSAR components developed will perfectly interface with the QT interface. Meanwhile, the data visualization should be of low latency to keep appropriate interaction for users. The hypothesis of this work is that, on a small-sized

automotive platform, for example, the TUCminiCar, an applicable and versatile solution may be implemented to fill the gap between the AUTOSAR data management facilities and the requirement for a dynamic, real-time visualization tool. In doing so, the proposal also tries to overcome the limitations of high-end, proprietary systems, besides offering an approach toward scalability for real-time vehicle status monitoring in research-oriented environments.

1.3 Motivation

Real-time monitoring and displaying of vehicle data have seen a great improvement in modern cars, complemented by advanced infotainment systems and digital dashboards. Equipped to deliver, these platforms make smooth and seamless transitions to critical updates on the performance of vehicles in real time, such as speed, engine performance, and sensor data. The outcome is an awesome user experience. These are highly inspired by AUTOSAR-a standard that allows modularity, scalability, and real-time communications among various ECUs of the vehicle. However, use of these advanced technologies is highly costly and thus proprietary, rendering imitation or modifications for research/academic purposes extremely hard.

In this respect, the TUCminiCar- a simplified automotive demonstrator, is very helpful in research and experimentation. Although it is less complex and technologically advanced compared to commercial vehicles, the TUCminiCar offers, in a simplified manner, the possibility of working on some of the most important topics in the automotive field, such as AUTOSAR integration, CAN bus communication, and real-time data visualization. Working in such an environment connects theory with practice and offers the best space for learning and testing new solutions.

Yet, the problem is that most solutions that can provide car status visualization for industries are proprietary, inflexible, and costly. Low-cost, customizable, and scalable solutions are still missing for real-time visualization of in-vehicle data, such as on the TUCminiCar platform. This project tries to fill this gap by integrating the AUTOSAR with the QT framework to provide an open, flexible system for real-time car status visualization that can also be implemented on smaller, research-oriented platforms.

This work will combine the standardized architecture of AUTOSAR with the powerful graphical capabilities of QT to realize an efficient Wi-Fi transmission of in-vehicle data and, simultaneously, real-time visualization of this data, with minimal latency and at high accuracy. The academic value of such a system is not confined to the system itself, since the methods and techniques developed can be transferred to full-

scale automotive systems and provide valuable insight into cost-effective, flexible, and scalable solutions for the automotive industry. The present work is an excellent learning opportunity within the TUCminiCar setup, contributing to the wide dissemination of advanced automotive technologies that will be increasingly more accessible and modifiable for further developments.

The thesis consists of six main chapters, starting with the Introduction, including the motivation, objectives, and background concerning the development of the AUTOSAR-compliant, QT-based dashboard for real-time data visualization. The Literature Review discusses existing technologies, standards, and research concerning vehicle data communication and visualization, such as main tools: CAN protocol and QT framework. The bases of the design choices for the system are set in this review.

The Concrete Design and Implementation chapter is dedicated to the concrete design and implementation of the gateway and visualization application, describing requirements, communication setups, and application structures to be set up for the realization of real-time data monitoring. The chapter entitled "Testing and Evaluation" ties testing and its results together, considering the arrangement and actual conduction of functional and performance tests, as well as practical tests on the TUCminiCar. For this reason, results will be discussed in terms of system responsiveness, accuracy, and effectiveness in real-time visualization. Finally, the Conclusion and Future Work chapter summarizes the achievements made by the project, reflecting on the advantages and limitations of the system. Further, it points out some suggestions for future improvement that might be tackled, insisting on the relevance of the system to current developments in the field of automotive data visualization.

2 State of Art

2.1 Evolution of Automotive User Interface

Now, in this section, let us discuss how far the evolution of a user interface has taken place in the automotive field. The evolution of User Interfaces within the domain of automotive expresses the wider transformation of vehicles from a purely mechanical to highly integrated digital platforms. In the early days, automotive user interfaces were rudimentary, with mainly analog dials, buttons, and levers for basic controls such as speed, fuel levels, and lighting. But with the rapid development of embedded systems, HMIs went way beyond simple knobs and buttons, now including complex digital displays, touch interfaces, voice commands, and connectivity services, turning today's vehicles not just much safer but also more user-friendly and engaging.

2.1.1 *Early Automotive User Interfaces*

Automotive User Interfaces in Early Days To date, the emphasis has been on displaying information to the driver that is quite essential. The majority of instrument clusters typically consist of analog speedometers, fuel gauges, and a limited number of warning lights. Early systems were completely mechanical, with very little interaction between the driver and the vehicle other than operating controls comprising the steering wheel, pedals, and gear shifter.

2.1.2 *Rise of Digital Displays and Infotainment*

Emergence of Digital Displays and Infotainment The transition to digital displays began in the 1980s, ushering in a new era in driver-vehicle interaction. It was also during this time that analog dials gave way to digital dashboards displaying information in more varied and dynamic ways. Along with other innovations, automotive infotainment systems began integrating audio, navigation, and multimedia features during the late 1990s and early 2000s. The touchscreen interface also gained popularity and made it possible to control all sorts of in-car entertainment, navigation, and in some cases even climate control, through a screen usually placed in the front. [18]

According to Schneegass et al. [19], this phase also brought more advanced user interface design methodologies. With the growth in the number of in-vehicle features, namely infotainment, driver assistance, and comfort systems, designers were confronted with the challenge of not overwhelming the driver with new interfaces or distracting him/her away from the main driving task.

The below diagram compares the interior of the Mercedes SL from the 1970s to the current Mercedes SL. This clearly shows how it has evolved.



Figure 2.1: Interior of Mercedes SL from 1970s compared to current model. [20]

Future Trends and Challenges

The key trends that have come up forth for the future of automotive UI are:

Higher Customization: The UI in the near future will be such that it provides maximum customization automatically to each driver based on his preference and pattern of usage. [21]

Voice and Gesture Control: With voice recognition continuously getting better, we should expect more cars to shift towards no-contact interaction modes. Voice interfaces, already replacing conventional buttons and dials, will be at the heart of ensuring that users interact with the vehicle without taking their eyes off the road. [22]

Immersive Interfaces: HUDs and AR interfaces project driving information onto the windshield today and will play an even greater role in the not-so-distant future. These, in turn, will allow for more intuitive interactions by making critical information appear in the line of sight for the driver, which will help reduce distraction. [23]

2.2 Current Industry Tools for Automotive HMI Design

Here's a table with some prominent **commercial software frameworks** used for creating automotive Human-Machine Interfaces (HMI).

Table 2.1: Overview of Commercial Software Frameworks for Automotive HMI Development

Framework	Company	Description
Qt Automotive Suite	Qt Group	A cross-platform Qt-enabled framework with a powerful means of developing UIs in automotive applications using Qt Quick, Qt 3D, and QML for real-time graphical interfaces. [24]
EB GUIDE	Elektrobit	Full-featured HMI development tool with support for multiple 2D and 3D graphics, used for Infotainment, Digital Clusters, and HUDs. [25]
Unity AutoTech	Unity Technologies	A real-time 3D development platform optimized for automotive applications, often used for 3D visualization, digital twins, and infotainment systems. [26]
GENIVI Development Platform (GDP)	GENIVI Alliance	Open-source collaborative framework for the creation of IVI systems; it provides, as such, the modular software to compose the UI and to integrate middleware. [27]
Kanzi	Rightware	A graphical user interface tool for designing interactive 2D and 3D automotive displays, ranging from digital instrument clusters up to infotainment. [28]
Altia Design	Altia	A cross-platform GUI development framework for embedded systems, focusing on high-performance HMIs for Automotive Dashboard and Infotainment Systems. [28]
QNX Platform for Digital Cockpits	BlackBerry QNX	An embedded platform is a digital cockpit solution built from operating system-level integration down to HMI development for automotive UIs. [29]
Crank Storyboard	Crank Software	It is an embedded HMI-focused GUI development tool, offering seamless design-to-deployment that really matters in automotive interfaces [30]
Android Automotive	Google	Android for automotive infotainment that can be customized and integrated with in-vehicle UIs by OEMs to embed Google services. [28]
CGI Studio	Continental AG	Automotive HMI Development Platform, proposed set of pre-built UI components, and rendering both for digital instrument clusters and infotainment. [31]



Figure 2.2: Instrumental Cluster designed by Kanzi. [32]



Figure 2.3: CGI based Instrumental cluster. [31]

2.3 CAN Bus in Infotainment Systems

In modern vehicles, more importance has recently been given to the use of CAN bus in integrating infotainment systems. The system controls multimedia, navigation, and connectivity services. These systems can also use the CAN protocol to communicate with other parts of the vehicle in a continuous flow of information regarding vehicle speed, fuel consumption, and other engine diagnostics, either to touchscreen displays or digital instrument panels with graphical user interfaces. Integration enhances driving comfort since information becomes smoothly available to both the driver and passengers. [33]

A very typical example of work done is by Rahul M. Patil et al., entitled "Infotainment System Using CAN Protocol and System on Module with Qt Application for Formula-Style Electric Vehicles," published in the year 2020 [34]. This research is focused on designing an in-car infotainment system as shown in the figure 2.4 below, that will read real-world data from a variety of sensors from the formula-style electric vehicle

and present that information to the driver. It logs data for post-race analysis that the engineer can use to infer performance based on vehicle safety. Key Components of the system includes:

System on Module: SoM performs the execution of data processing because it is small in size and performs well during operation.

CAN Protocol: CAN protocol has been used for transmitting data in real time among sensors and infotainment.

Qt Framework: It does the work of showing the data in a user-friendly interface manner so that the driver and the crew could access it easily.

The whole system was tested in workshop and on-track conditions, hence proving a very robust means of real-time solution for vehicle dynamics monitoring. The success of this system outlines the value of CAN bus in intra-vehicular communication for electric and performance vehicles effectively.



Figure 2.4: GUI that appears on the Dashboard. [34]

2.4 Overview of Qt in Embedded Systems Development

Qt framework has become highly important in embedded systems development, especially within the automotive industry, due to its cross-platform capabilities and its highly flexible and efficient GUI. The success of Qt in embedded systems has been mainly because Qt can abstract the underlying hardware effectively, thus providing an environment to the developers for designing rich UIs that can execute on many platforms with minimal changes in code. In particular, it becomes an enabler in the automotive sector, while integrated infotainment systems, instrument clusters, and ADAS require all real-time performances that are truly dependable and easy to use.

2.4.1 Qt for Automotive UI's

Qt has been used everywhere in automotive development, from IVI systems to ADAS. The rich toolsets that Qt provides, such as Qt Quick, Qt Widgets, and Qt 3D, make life easier for developers to design and deploy really interactive, responsive UIs. Most of these systems need real-time data display coming from various sensors in the vehicle; Qt efficiently manages this using the Qt Model/View architecture, seamlessly integrating the backend data with the frontend graphical elements.

For instance, the adaptiveness of the Qt framework was shown by Hasan Aljaere in his 2021 master's thesis, while realizing an adaptive user interface for automotive demonstrators. Targeting the execution on a Raspberry Pi, it provided a user interface to interact with various components running on vehicle sensors and cameras in real time. Qt's flexibility allowed for a tenable UI that could handle runtime changes without having to recompile the system—a big plus when considering Qt for real-time automotive applications. The Qt5 Home page, styled as a BMW I series tachometer, showed car data and warnings using dummy values in the BlackPerl demonstrator, fixed settings being immutable. [35]



Figure 2.5: Black Pearl Display Home. [35]

Another example of the relevance of Qt in automotive embedded systems is the thesis work of Bramastyo Harimukti Santoso, a master's student who developed, in the year 2020, a 3D dynamic user interface framework for vehicles. His work underlined the usage of Qt for the development of an appealingly dynamic interface for car dashboards, which also will be driven by real-time inputs from sensors installed in the vehicle. This framework developed the flexibility of Qt to handle instrument cluster 3D graphics, thereby improving driver experience and interaction with the vehicle. [36] The diagram below represents the simulation to change the car colors in 3D designed by QT components.

Some projects emphasize using open-source software and inexpensive hardware to provide low-cost infotainment systems. The "In Vehicle Infotainment Demonstrator" by Sanell and Samuelsson [37] discussed a CAN-based infotainment system as illustrated in the figure 2.7 using one multi-purpose computer module with Linux OS for developing a low-cost demonstrator. This indeed shows the potential of open-source tools and off-the-shelf hardware in automotive applications also. In their system, it provides simple infotainment like control audio or navigation.

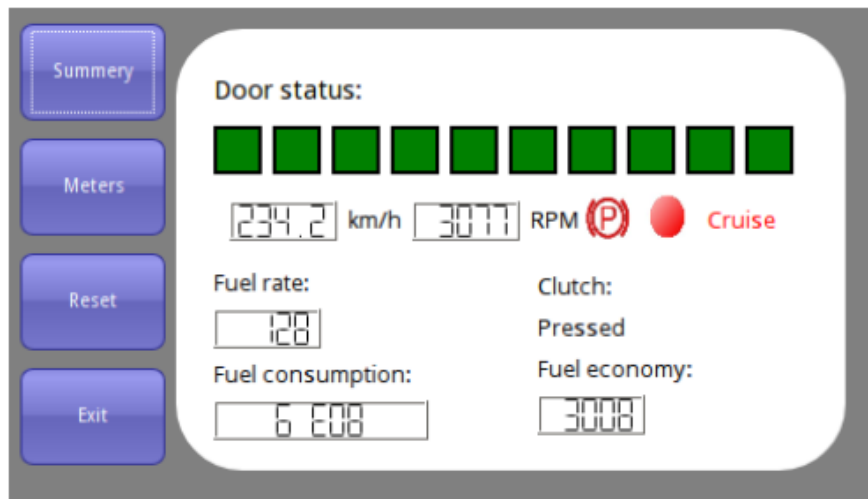


Figure 2.6: View of the Demonstrator. [36]

2.4.2 Qt in Embedded Linux Systems

One of the strong points of Qt is its great integration with Embedded Linux, which, besides being a commercial framework licensed under GPL, makes it a go-to solution in automotive because robustness and efficiency in resource usage are considered very vital. Qt provides deep support for multithreading, cross-platform compatibility, and highly optimized graphical rendering—all characteristics basically requisite for automotive UIs to get going reliably in resource-limited computing environments.

One such example was Qt Quick used on an LCD display instead of regular analog instrumentation clusters for automotive purposes by Yan-jun Di et al. The move to LCD faced not only a face uplift but also allowed dynamic rendering of graphical representations of speedometers, fuel gauges, and other real-time data indicators inside the instrument cluster. Qt's modular architecture allowed great developers to come up with scalable cross-platform solutions across different car models. The beauty of this adaptability is that it cuts development time, reduces the cost; basically, the same software can be easily deployed on different hardware configurations without extensive modification. [38]

Qt/Embedded has gained popularity in several embedded applications due to flexibility and efficiency, running in real time and involving direct interaction by the user with the complicated systems. Liu-Yang [39] considered a research work on the use of Qt/Embedded in intelligent management systems. In this research work, Qt/Embedded was used to implement a graphical user interface in an interaction with information-intensive applications. Qt was good to go with such an application, as it was proficient in supporting multitasking along with real-time graphical updates, hence working nicely when there was a need to process complex data without compromising a responsive and user-friendly interface.

This study is of prime importance for its future application in embedded automotive systems, which need trustworthy interfaces since several embedded applications may be handled at one given time. For instance, in an advanced in-car infotainment or even driver information display, software would be required to deal with navigation data, media control, and live vehicle diagnostics-computations that should not stall. Qt/Embedded can do this because it is adaptable and efficient with memory, and thus will be easily implemented into HUDs and digital instrument clusters to ensure latency is little to nonexistent to stay on pace with driver safety and satisfaction. Qt's modular development offers several opportunities for adding or modifying components without requiring an overhaul of the whole system and can support over-the-air updates as usual for connected vehicles.

2.5 AUTOSAR and QT based Visualization

Håkan Lönn's "Project within Vehicle Development: Report on AUTOSAR and QT-Based Visualization" [40] is dedicated to integrating AUTOSAR with QT in the development of the real-time vehicle visualization system. This project therefore shows that standardized modular components within the scope of AUTOSAR are perfectly able to communicate within a vehicle system and hence form the basis for scalable and flexible automotive software. This work was supposed to involve the development of a GUI interface using QT, where all data from the different AUTOSAR modules would be mapped and then served in real time to the end user, so that all vehicle parameters are dynamically monitored. Its specific particularities include the using of CAD models in the QT interface for three-dimensional visualization of vehicle parts. This option thus allows not only an interactive highlighting of a desired part of the vehicle but also for its monitoring, which improves situation awareness through complex data visually and intuitively presented. Visualization provides real changes and alerts on the status of components in real-time to help diagnostics and the users get better insights into the status of vehicles.

It was also heavily tested for verification-that with as little latency as possible, real updates of data could be handled; a factor extremely important for applications based on timely information. The system showed the compatibility of AUTOSAR with QT, having flexibility in various integrations of automotive environments, even such advanced systems as driver assistance or autonomous ones.

While the work with the TUCminiCar applies the very same combination of AUTOSAR-QT, it focuses on an affordable 2D visualization for educational purposes. While the Lönn project was a high-detailed 3D visualization toward a more immersive experience, your approach used QT functionality in a simpler form suitable for easily accessible real-time monitoring. The contrast of such a setup underlines the flexibility of the AUTOSAR-QT setup: being capable of scaling down to simple educational applications, yet easily scalable up to complex and industry-grade visualization systems with added interactivity. The inclusions like CAD-based visualizations or even interactive 3D models used in Lönn's project would be a lot more detailed and appealing in your setup, especially if expanded for applications beyond research.

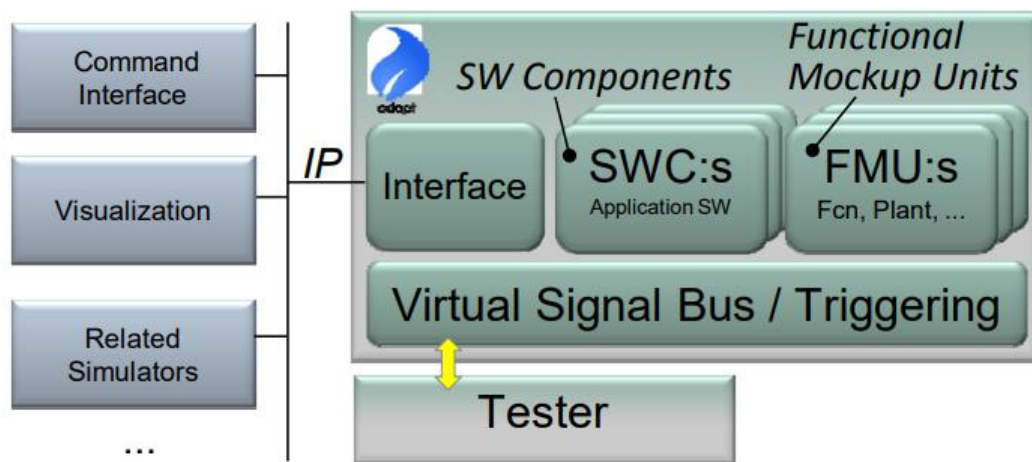


Figure 2.7: Simulation Platform. [40]

It is, after all, based on two of the most well-liked automotive UI enhancements: using CAN protocol and Qt framework for real-time data visualization. However, it targets the TUCminiCar, putting much effort into making the hardware affordable and accessible using Raspberry Pi hardware. That's what makes it different from the existing solution. It uniquely features dual-sensing data integration, where switching between distance sensors and LiDAR occurs dependent on speed, while others leverage UDP for a speed-optimized way of sending data. It targets educational and research applications, providing an inexpensive, versatile platform that can be used to run real automotive interface experiments.

3 System Architecture

This section describes the architecture of the TUCminiCar system, including the hardware and software components relevant to the thesis. It provides for real-time acquisition, processing, and visualization by means of an embedded system compliant with AUTOSAR. The subsections go on to describe the hardware elements of the Aurix microcontroller and Raspberry Pi, along with the CE Box, which manages the key functions of the vehicle. It further describes the software architecture, focusing on how the AUTOSAR components interact with hardware to support CAN communication in facilitating data exchange between system components. System integration will be discussed along with a Qt app for real-time visualization of vehicle status.

3.1 Overview of the TUCminiCar

The TUCminiCar is an educational and research demonstrator that has been developed by the Department of Computer Science and Engineering, TU Chemnitz. It serves as a model for promoting research studies in automotive embedded systems, especially concerning the integration of AUTOSAR components. This system is intended to emulate modern automotive system features and allow students and researchers to work practically on real-world automotive technologies in a controlled environment. The TUCminiCar uses a hardware-software integrated system, specially designed for collection, processing, and visualization in real time. The Aurix TC387 microcontroller forms the central unit of this system, entrusted with the main task of vehicle control and interfacing with other units of the system. In addition, the car is equipped with a suite of sensors, such as LiDAR and ultrasonic sensors, that continuously scan the surroundings and update data in real time.

In the system, data bound over the CAN bus is captured through a Raspberry Pi interfaced with the PiCAN2 module and sent wirelessly via Wi-Fi. This real-time data is taken from the CAN bus, which plays a major part in the monitoring of operational parameters of the vehicle and visualization of status updates. The PiCAN2 module provides CAN-Bus functionality for the Raspberry Pi. This system has a Central Control Unit, CE Box, hosting a Raspberry Pi connected with a display for real-time visualization of the status of the demonstrator. It shows in real time data transmitted over Wi-Fi from the TUCminiCar. A Qt application handles and displays in real time data, which are processed from the sensors and actuators of the vehicle.

The Figure 3.1 below shows the overall structure of the TUCminiCar. It is designed in such a way that it is supposed to illustrate data flow between the Aurix microcontroller, sensors, Raspberry Pi, and the CE Box. The design also shows how the compliance of the AUTOSAR standards could be achieved through the integration of embedded hardware and software to make modern automotive architectures easier for students and researchers to study and explore.

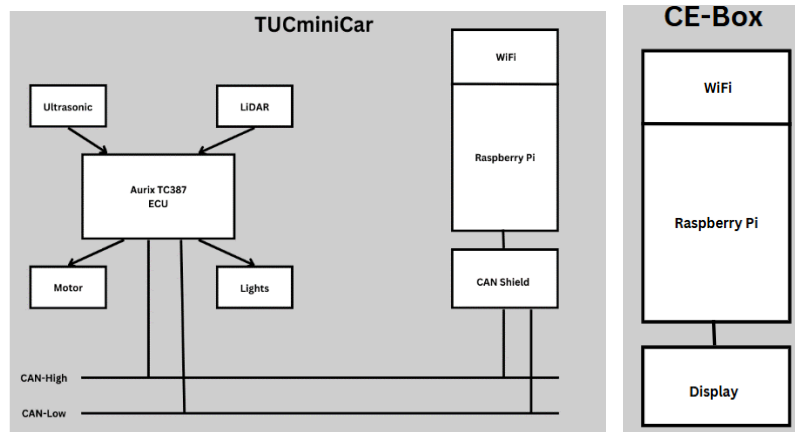


Figure 3.1: Overview of the TUCminiCar.



Figure 3.2: Car Demonstrator- TUCminiCar.

3.2 Hardware Components

3.2.1 Aurix Microcontroller

Infineon Technologies' microcontroller, the Aurix TC387 [41], is a high-performance and 32-bit TriCore-based microcontroller designed with several automotive applications in mind, including elaborate automotive system management such as powertrain control, vehicle safety systems, and features around autonomous driving. The Aurix microcontroller plays an important part in the TUCminiCar-a core controller that handles communication between the Controller Area Network (CAN) bus, sensors, and actuators.

The multi-core processing enabled by the Aurix TC3xx family supports up to six cores running at 300 MHz, thus enabling the efficient handling of complex tasks in real-time. Real-time-optimized architecture, including lockstep cores for safety-critical functions, conforms to the highest classification, ISO 26262 ASIL-D, within the automotive industry. This makes it perfect for use cases that require high reliability with minimum latency, such as vehicle control and ADAS.

Along with safety, the hardware security modules that are integrated within the Aurix microcontroller are meant to support preventing unauthorized access, tampering, and theft of intellectual property. These modules ensure secure communication, which is absolutely necessary in today's connected automotive domain. In addition to that, the microcontroller allows various communications: CAN FD, FlexRay, Ethernet, and LIN, ensuring compatibility with the largest range of vehicle subsystems.

In such applications, like the TUCminiCar, it efficiently controls actuators, such as motors, lights, and servo mechanisms, where input is given by a wide range of sensors, including ultrasonic sensors and LiDAR. This is processed on the Aurix TC387, with CAN bus transmitting data to other system components for further processing and visualization; this includes the Raspberry Pi. The versatility and performance of the Aurix microcontroller makes it a basic pillar of the TUCminiCar architecture, enabling seamless real-time control and interfacing with the rest of the system.

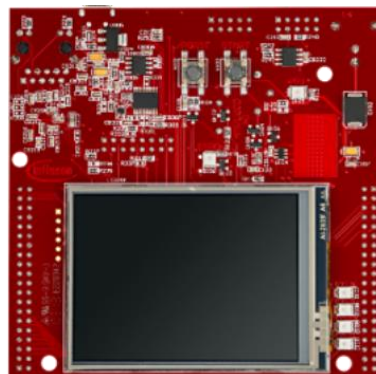


Figure 3.3:AURIX™ TC387. [41]

3.2.2 Raspberry Pi and Pican2

Although it is very small in size, the Raspberry Pi 3B+ [42] is a pretty capable single-board computer, undertaking several tasks in the TUCminiCar system. It acts as an interface in capturing CAN bus data for wireless transmission. The Raspberry Pi is running on the latest version of Raspbian OS and has been interfaced with the PiCAN2 [43] HAT (Hardware Attached on Top) to achieve CAN communication

among different ECUs available in the vehicle. The PiCAN2 module integrates the MCP2515 CAN controller and the MCP2551 CAN transceiver to provide a complete function for CAN 2.0B, thus supporting high-speed data transmission. This CAN module, PiCAN2, allows connecting the Raspberry Pi to the CAN bus, which forms part of many automotive systems; these need real-time interaction between an Aurix microcontroller with other vehicle subsystems.

The Raspberry Pi will implement a QT-based visualization system in this project. It will be a central processing unit that will get the data from the AUTOSAR software component via Wi-Fi and show its real-time values on the QT interface. Presently, there are quite a number of research studies that involve the use of Raspberry Pi integrated with vehicle networks-such as a Controller Area Network-to show that it's one of the viable solutions for real-time data processing and monitoring in automotive applications.

It makes it possible to collect and send data from different sensors, such as LiDAR and ultrasonic devices, combined with actuators like motors and lighting systems using CAN-High and CAN-Low signal channels. The use of a SocketCAN driver compatible with Raspbian enables the Raspberry Pi to communicate with the CAN network with ease and directly observe the functionality of the vehicle with real-time diagnostic assessment. Moreover, the data transferred from Raspberry Pi 3B+ through Wi-Fi is sent to the CE Box or the central display unit in which the Qt application displays the status of the vehicle. Thus, the interfacing of CAN bus communication with Raspberry Pi and PiCAN2 provides a low-cost solution that may have much scalability potential for automotive projects needing robust real-time management and visualization of data.

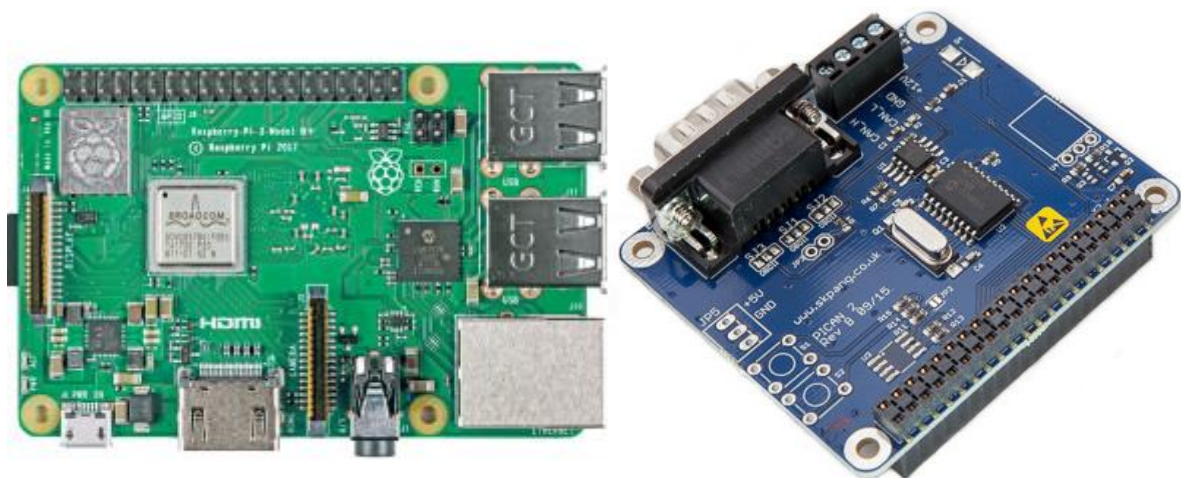


Figure 3.4: Raspberry Pi 3b+ and PiCAN2. [42] [43]

3.2.3 Sensors Used in the TUC Mini Car

Ultrasonic Sensor (HRLV-ShortRange-EZ)

The HRLV-ShortRange-EZ ultrasonic sensor [44] is used in the TUC Mini Car for proximity measurement. It plays a critical role in detecting obstacles around the vehicle for close-range monitoring. These sensors are very useful for applications requiring precise short-distance measurement.

Measurement Range: 2 cm to 500 cm.

Accuracy: ± 3 mm, ensuring reliable readings in close proximity.

Response Time: 100 ms, suitable for real-time applications.

Usage in the System:

- Placed strategically around the vehicle to provide accurate distance measurement data.
- Visualized on the dashboard with a color-coded safety indicator for immediate interpretation by users (red, yellow, green).

Advantages:

- High accuracy for short-range applications.
- Compact and easy to integrate with the existing system.

Limitations:

- Limited to 500 cm, which may not be sufficient for high-speed scenarios.



Figure 3.5: Ultrasonic Sensor (HRLV-ShortRange-EZ). [44]

LiDAR Sensor (LD19)

The LD19 LiDAR sensor [45] supplements the ultrasonic sensors, providing a 360-degree environmental map for the TUC Mini Car. In particular, it is effective at long-range detections and for drawing a detailed map of the surrounding environment.

Operation Range: 2-1200 cm

Resolution: Accuracy ± 45 mm, suitable for detecting obstacles with greater accuracy at further distance

View Angle: 360°, Full scan

Frequency: 10 Hz-100 ms/Scan, for timely response.

Usage in System:

- Mounted to provide an overhead view of the surroundings of the vehicle.

- Data from the LiDAR sensor is displayed on the dashboard for velocities above 60 cm/s for users to see how close each obstacle is in all directions.

Advantages:

- Large scanning range and high angular resolution
- Good for dynamic environments, where multiple objects have to be tracked.

Limitations:

- The accuracy is slightly less for very near objects as compared to ultrasonic sensors.
- Computationally intensive data processing due to large volumes of data generated.



Figure 3.6: LiDAR Sensor (LD19). [45]

3.2.4 CE-Box

The CE box does form the backbone of the TUCminiCar for it is the main platform for data visualization and real-time monitoring. It contains a Raspberry Pi 3B+, whose job is to run a Qt-based application that does data visualization related to the vehicle and a PiCAN2 module. Although this implementation of the thesis doesn't use CAN communication very much, the PiCAN2 remains one of the hardware configurations for possible future scalabilities and integrations with other sub-systems through CAN. The main unit of processing in the CE Box is the Raspberry Pi, which runs the Qt application responsible for collecting real-time data from the vehicle, processing, and then presenting it on a connected display.

The CE box is designed to wirelessly transmit data over Wi-Fi from the vehicle sensors and an Aurix microcontroller to the display interface, which visualizes all critical metrics of the vehicle-speed, system health, and sensor outputs. The Qt-based application becomes the foremost software in user interaction and monitoring. While the functionality of CAN bus through a module called PiCAN2 may not be required with this implementation, the hardware CE Box retains this to increase scope in the future where direct CAN communication might be required with other subsystems or research projects.

The CE Box front panel as seen below is built with various connectivity options, such as USB, Ethernet, and power supplies, ensuring that the system is comprehensive enough to withstand or support robustness in research and operational undertakings. Other inclusions are LED indicators on the front panel that provide appropriate feedback about the runtime status of an operating system. [46]

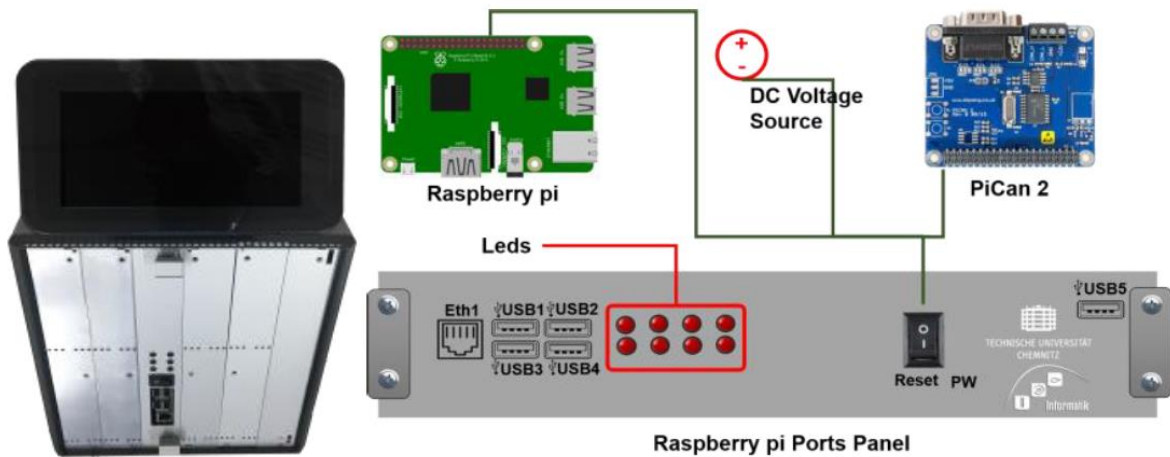


Figure 3.7: The CE-Box and the front panel connection. [46]

3.3 Software Components

3.3.1 Autosar Software Components of TUCMiniCar

The TUCminiCar shall contribute to the AUTOSAR-compliant environment in order to enable a module-oriented development of software components for automotive applications. AUTOSAR stands for Automotive Open System Architecture, an open standard software architecture that allows for efficient integration-software and hardware modules-across various control units within a vehicle, thus enabling compatibility, scalability, and ease of maintenance. This is done to provide a preconfigured environment for AUTOSAR-related developments in which students and researchers can easily implement and extend existing functionalities.

Several SWCs that have already been implemented in the TUCminiCar's AUTOSAR BSW setup will be reused in the scope of this thesis. Their utilization in acquiring sensor data from the vehicle and transmitting it for visualization by means of a Qt-based application over Wi-Fi will be elaborated. Relevant SWCs are introduced in the following:

InputSonar: This component reads in the data captured from the vehicle's ultrasonic sensors. The module captures the measurement of distance and converts it to a useable variable for further processing. First, these sensor values are sent as CAN

messages that later are forwarded over Wi-Fi for visualization inside the Qt application.

SensActInputLiDAR: This SWC processes the data coming from the LiDAR sensor to detect the distance of objects in the close environment of the vehicle. The LiDAR data becomes very important in mapping the real world with a view to efficient and active visualization of the vehicle through the combination of other sensor data.

Communication Manager: This, in turn, is responsible for transmitting critical vehicle status information throughout the system. This includes the status of the steering wheel, engine status, car speed, battery voltage, and lights and blinkers.

AUTOSAR SWCs below are responsible for ensuring data acquisition from sensors is processed by the Aurix microcontroller and transmitted over CAN. The CAN messages are intercepted through the PiCAN2 module coupled with the Raspberry Pi in the CE Box and sent, via Wi-Fi, to the Qt-based visualization application for real-time availability of the vehicle's operational data. These data are then graphically rendered by the Qt application, showing the vehicle's current status regarding speed, sensors, and control inputs.

This diagram below illustrates the flow of inputs and outputs across various software components in TUCminiCar, all from AUTOSAR. It also explains on how the SWCs process sensor inputs, including sonar, encoders, and LiDAR, to generate commands for actuators like lights, steering, and motor control, and how everything interchanges information using the CAN bus. This combination of standardized software architecture AUTOSAR with Qt-based visualization can now empower researchers and testers to conduct powerful research and tests into automotive embedded systems. This architecture will be able to support the development of reliable, modular software components that are easily scalable and adaptable to the variable hardware environments within the system of the TUCminiCar.

Message Identifier: It describes the purpose of the message and the priority of that vehicle network.

Data Length Code (DLC): This is a very simple thing which is the number of bytes in the data field. Normally 0-8.

Data Field: This represents the actual data that is being sent out. For example, rate of movement, steering characteristics among much other sensing components.

CRC (Cyclic Redundancy Check): Checking the errors.

ACK (Acknowledgment Slot): It serves the purpose of confirming successful reception of received message by the target.

For the clarity of what has been explained so far, let us take an instance. In TUCminiCar, there is a CS1 message which pertains specifically to vehicle communication, and it is a very interesting message. This message is vital for understanding the status of vehicle parameters such as steering, status of car engine, car speed, the level of the battery charge and the condition of lights containing CS1. It is sent regularly every 100 milliseconds via the bus in order that all the components of the system can have the actual parameters of the vehicle at any time.

Message Structure and Fields includes:

Message Code: CS1 (Car Status 1)

Message CAN ID: 0x06B

DLC (Data Length Code): 8 bytes, indicating the message contains 8 bytes of data.

Period: 100ms, meaning the message is sent every 100 milliseconds.

Timeout: 500ms, after which the message is considered stale if not received.

Description: Provides the status of the vehicle's steering, speed, battery voltage, and light statuses.

Each byte in the message corresponds to the listed parameters. These are the details for the bytes D0-D7:

D0 (Steering Status): This contains the status of the steering-1 bit per unit in resolution-offsetting by 100. Therefore, it supplies real-time data about the status of the vehicle's steering system: cantered, left, or right.

D1 (Steering Feedback - Potentiometer Feedback): This contains real-time data on steering feedback; the resolution is 0.02 V/bit. It feeds from a potentiometer tracking the actual position of the steering wheel.

D2 (Engine Status): Indication of the on/off state of the engine, given that 1% change per bit gives a resolution, offset by 100.

D3- D4 (Car Speed): Represent car speed in mm / s with the resolution of 1 mm / s per bit ranging from 0 as minimum to 1200 as maximum. In this way, the system is immediately able to monitor and immediately control the speed of a vehicle.

D5 (Battery Voltage): It returns the current battery voltage with 0.1 V/bit resolution, which is useful to monitor health and charge status regarding the vehicle's electrical system.

D6-D7 (Light Statuses): These bytes represent the on/off status of several in-vehicle lights: for example, blinkers, brake, reverse, and park lights. Examples: high beam and low beam will be controlled by Boolean values ON = 1, OFF = 0. Below is the CAN table representation for the Car Status with a CAN message ID: 0x6B.

D0								D1								D2								D3								D4							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Steering Status (Resolution: 1%/bit) (Offset: 100)								Steering Potentiometer Feedback (Resolution: 0,02 V/bit)								Engine Status (Resolution: 1%/bit) (Offset: 100)								Car Speed (mm/s) (Resolution: 1 mm/bit)															
0x00: 100% Right ... 0x63: 1% Right 0x64: Neutral 0x65: 1% Left ... 0xCB: 100% Left 0xCD..0xFE: Reserved 0xFF: Not Available								0x00: 0.00V ... 0xFE: 5.08V 0xFF: Not Available								0x00: 100% Reverse ... 0x6B: 1% Reverse 0x6C: Neutral 0x65: 1% Forward 0xCB: 100% Forward 0xCD..0xFE: Reserved 0xFF: Not Available								0x0000: 0 mm/s ... 0x0080: 1200 mm/s 0x00B1: Reserved 0xFFFE: Reserved 0xFFFF: Not Available															

Figure 3.9: CAN Message table for CS1 from D0-D4.

D5								D6								D7									
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
Battery Voltage (Resolution: 0,1 V/bit)								Left Signal Status		Right Signal Status		Low Beam Status		High Beam Status		Brake Lights Status		Reverse Lights Status		Park Lights Status		Reserved			
0x00: 0.0 V 0x01: 0.1 V ... 0xFE: 25.4 V 0xFF: Not available								0b00: OFF 0b01: ON 0b10: Reserved 0b11: Not Available		0b00: OFF 0b01: ON 0b10: Reserved 0b11: Not Available		0b00: OFF 0b01: ON 0b10: Reserved 0b11: Not Available		0b00: OFF 0b01: ON 0b10: Reserved 0b11: Not Available		0b00: OFF 0b01: ON 0b10: Reserved 0b11: Not Available		0b00: OFF 0b01: ON 0b10: Reserved 0b11: Not Available		0b00: OFF 0b01: ON 0b10: Reserved 0b11: Not Available		0b00: OFF 0b01: ON 0b10: Reserved 0b11: Not Available		0b11: Not Available	

Figure 3.10: CAN message table for CS1 from D5-D7.

This message is sent as a standard message, with the format according to CAN 2.0B. Each signal in the message has its types, resolutions, and offsets in bits. The signals are then packed into this 8-byte message, each field representing another vehicle parameter that can be encoded within one message.

Data Encoding: Each of the different data is encoded in one or more bits in the message. For example, car speed in mm/s is encoded into D3 and D4 using a scale factor of 1 mm/s per bit. That is, every bit composing the byte D3 expresses an extra mm/s of speed. Message interpretation: The CAN bus sends messages to different system components. According to predefined message identifiers (IDs), after receiving this message, it extracts data by decoding individual signals based on scaling factors provided in the CAN database-for example, speed and battery voltage.

The entity receiving the CS1 message, for example, will interpret some steering status by parsing out the appropriate bits from D0 into a value significant to it, such as left, center, and right, and then display or act on that information. Likewise, the value of D5 will be utilized to convey real-time battery voltage. Thus, this setup allows for real-time continuous monitoring and updating of the current state of the vehicle, hence allowing its effective control and visualization in systems like Qt-based interfaces used in the visualization of TUCminiCar's operations. Like the same way below is the representation for the Distance Sensor (DS1 and DS2) **and** LiDAR Sensor (LS1 to LS5) CAN message formats.

Distance Sensor (DS1 and DS2) CAN Message Format

Message Structure and Fields:

Message Code: DS1 (Distance Sensor 1), DS2 (Distance Sensor 2)

Message CAN ID: 0x60 (DS1), 0x61 (DS2)

DLC (Data Length Code]: 8 bytes

Period: 100ms

Timeout: 500ms

Description: Provides the measured distance for up to 10 sensors connected to the car.

Byte Details: Each byte provides information about one sensor whose values from 0x00 to 0xFA in the resolution with 1cm/bit from 0cm to 250cm.

LiDAR Sensor (LS1 to LS5) CAN Message Format

Message Code: LS1 (LiDAR Sensor 1), LS2, LS3, LS4, LS5

Message CAN ID: 0x62 to 0x66 (for each LiDAR Sensor)

DLC (Data Length Code): 8 bytes

Period: 100ms

Timeout: 500ms

Description: Each byte provides information about values in 10-degree intervals, ranging from 0x00 to 0xFA, with a resolution of 1 cm/bit, representing distances from 0 cm to 250 cm.

3.3.3 Data Flow and Integration with QT Application

One of the key factors in TUC Mini Car architecture is real-time data flow, which enables smooth interaction of hardware components and the GUI used for vehicle status monitoring. The combination of CAN communication with UDP transmission and Qt-based visualization is used here to offer low latency, so a user-friendly interface is developed. Further in the paper, we are going to give in detail all the components and steps in the real flow.

CAN Data Collection and Preliminary Processing: The core of the system is a Raspberry Pi acting as a gateway that is equipped with an Aurix microcontroller and a PiCAN2 module. The PiCAN2 extends the capability of the Raspberry Pi to interface via the CAN bus, the backbone for communication in most parts of the TUC Mini Car, by several sensors and software components.

CAN Bus Communication: CAN is continuously broadcasting the data the sensors of the vehicle have-an ultrasonic, LiDAR, and control units via predefined structures of the frames of CAN messages. In other words, a Qt-based application is deployed in the Raspberry Pi and is supported by the QCanBus library for interaction with the CAN. It provides an efficient way to configure, connect to, and retrieve CAN frames using a high-level API.

Reading CAN Frames: This application continuously reads incoming frames from the PiCAN2 module and identifies them through their frame IDs, extracting data out of them for further processing. **Data Extraction and Formatting** Once the CAN frames are received, the system will then process the data for visualization:

Frame Identification: Each CAN frame has a frame ID that identifies its type of data, such as speed, battery voltage, or steering angle.

Parsing of Payload: The payloads contain the sensor values encoded and have to be parsed into a well-structured byte array called `udpPayload` suitable for transmission.

Catching Bad Data in Payload: In addition, it performs a check in case there is some bad or missing data inside the payload. If this were the case, it triggers fallbacks on items such as visual warnings for sensor statuses when unavailable, showing defaults.

Wi-Fi UDP Transmission: The processed CAN data is then sent over to the CE Box using Wi-Fi with the UDP protocol.

Advantages of UDP: The use of UDP ensures low latency in transmission, which is very important in real-time applications. Unlike TCP, UDP avoids handshaking and retransmission overheads, making it faster for continuous streams of data.

Packet Creation: The `udpPayload`, containing the CAN frame ID and payload, gets encapsulated into a UDP packet. This is then addressed to the Raspberry Pi of the CE Box with an appropriate predefined IP address and port configuration.

Transmission: Using `QUdpSocket`, the application sends these packets over Wi-Fi. The system is designed to ensure that packets are sent out at regular intervals equal to the CAN message period to keep synchronization.

The Qt-based application running on the CE Box processes incoming UDP packets and updates the user interface in real-time as follows:

Receiving UDP Packets: Using QUdpSocket, the application listens for incoming packets. The packets are decoded upon receipt to extract the original CAN frame ID and payload.

Data Decoding and Mapping: The extracted data is decoded using the predefined definitions of CAN messages.

Each data field, for example, is mapped onto its corresponding UI element on the dashboard, for instance, speed or battery voltage. Real-Time Updates: Qt application updates runtime to reflect the latest status of the vehicle. Examples include updating changes in speed on the speedometer and on the battery indicator, the change in the battery level.

User Interface Design and Visualization: The Qt Quick and QML-based dashboard application is developed to enable an intuitive, user-friendly interface.

Visual Components: The following are the sliders, gauges, and indicators that make up the interface to show some key parameters of speed, battery voltage, and proximity sensor data.

Dynamic Visualization: Proximity information obtained from ultrasonic and LiDAR sensors will be represented by color-coded regions around the car icon, representing safe, moderate, and danger zones.

Status Indicators: The color and state of status indicators such as lights and blinkers would change according to real-time data. Error Alerts: The interface also handles errors by showing warning messages or changing component colors to raise alerts to the user about some kind of malfunction. Low-Latency Updates: Because CAN and UDP are combined in this system, it cuts down the delay between data generation and its visualization, thereby enhancing real-time monitoring.

Scability: The modular design of the whole architecture allows adding/extending it with other sensors and functionalities that will be integrated in the near future without major changes; Error Tolerance: Documents missing or invalid, not yielding any error, would pop up a notification while keeping the big functionality running in most of the cases.

It means a high-level solution flow by incorporating CAN communication, UDP transmission, and Qt visualization for monitoring and interfacing with the TUC Mini Car. It uses industrial-strength protocols and tools; therefore, this architecture is at least dependable and scalable, while making a valuable contribution in research and education for automotive industries.

4 Design and Implementation

In this chapter the structure of the qt projects that is developed will be discussed along with its design and implementation respectively.

4.1 System Requirements

While the detailed breakdown has been given in the preceding sections, this section will give an account of specific hardware components that were used in the actual implementation. Here, each component is described by means of its specification and role it plays in the TUCminiCar system, with particular attention to their contribution to the whole setup in terms of data acquisition, processing, and visualization.

4.1.1 Hardware Requirements

Here's a table summarizing the Hardware Requirements for the TUCminiCar System implementation, including each component's name, specifications, and function in the setup. For detailed explanations on the individual components refer to section 3.2.

Table 4.1: Hardware Requirements.

Name	Specification	Function
AURIX™TC387 Microcontroller	TriCore™-based 32-bit MCU, TC387	Primary control unit for real-time processing of vehicle data and CAN communication
Raspberry Pi 3B+	Broadcom BCM2837, 1.2 GHz Quad-Core ARM Cortex-A53, 16GB SD Card	Runs the gateway application, processes CAN data, and transmits it via Wi-Fi.
PiCAN2 Hardware	MCP2515 CAN controller with MCP2551 transceiver	Interfaces with the CAN bus on the Raspberry Pi, enabling CAN data acquisition
Ultrasonic Sensor (HRLV-ShortRange-EZ)	Range: 0-5m, Resolution: 1mm	Provides short-range distance measurements for detecting nearby objects around the vehicle
Lidar Sensor (LD19)	Range: 0-12m, 360° scanning	Captures environmental data with a 360° field, enabling object detection and navigation
CE Box with Display	Resolution: 800x400, 7-inch and Raspberry Pi 3B+	Acts as the central control unit for visualizing real-time data with the Qt-based application

This table provides a structured overview of the specific hardware components essential for implementing the TUCminiCar system. Each item plays a distinct role in data acquisition, processing, communication, and visualization.

4.1.2 Software Requirements

Here's a table summarizing the Software Requirements for the TUCminiCar System implementation, detailing each software component's name, specification, and function in the setup:

Table 4.2: Hardware Requirements.

Name	Specification	Function
Linux OS	Latest version, preferably Ubuntu	Used on a laptop for development, flashing the Raspberry Pi, and testing with virtual CAN
Raspbian OS	Latest release for Raspberry Pi 3B+	Primary operating system on the Raspberry Pi, supporting CAN data collection and UDP transmission and QT visualization
Qt 5.15	Compatible with Raspberry Pi 3B+	Provides the framework for developing the Qt-based visualization application for the CE Box
Networking Libraries	QCanBus and QUdpSocket in Qt framework	Enable CAN communication and UDP data transmission between Raspberry Pi and CE Box

This table organizes the essential software components, specifying each one's purpose and role in developing, running, and managing the TUCminiCar system. Now since the whole application is developed in Qt 5.15, let us dive deep into the elements used in the QT and their respective application.

4.2 System Design Overview

Based on the basic knowledge of automotive systems, CAN communication, and AUTOSAR components described in previous chapters, this section will investigate in-depth system design for a TUCminiCar data flow and visualization setup. Here, we present the realization of two different Qt applications: one is the Gateway Application, running on Raspberry Pi, responsible for collecting and transmitting data, and another-the Dashboard Application-operating on the CE Box for visualization of real-time data of the vehicle.

It includes data acquisition by sensors through CAN communication, UDP transmission in inter-device data transfer, and real-time visualization on the

dashboard application. The information to be visualized is developed according to the provided sensor and CAN messages to state the vehicle's status correctly. The setup is thus done in a way that the user will be able to view critical metrics of the vehicle under normal operating conditions and at edge conditions like unavailability or signal errors of sensors.

Below is a summary of each of the implemented features, including the type of data visualized and the expected outcome. It follows a structured approach, giving insight into how the real-world information flows and, at the same time, showing the capacity of the system in dealing with different scenarios-with improved functionality and user experience.

Table 4.3: Requirements of the Qt Visualization.

Feature	Type of Data Visualized	Expected Outcome
Steering Status	Direction: Right, Neutral, Left	Visual indication of steering direction, with default neutral
Engine Status	Gear: Reverse, Neutral, forward	Displayed as "N" for neutral, "D" for forward, and "R" for reverse, with numeric feedback (e.g., forward = positive %). Unavailable shows "--%".
Car Speed	Speed (0 mm/s - 1200 mm/s)	Displayed in 0-120 cm/s range with progress bar. Unavailable status displays "--".
Battery Voltage	Voltage (0.0V - 25.4V)	Displays battery % with low-battery warning (<20%) in red, and critically low (<5%) with blinking and shutdown alert at 0% and then the application shuts down.
Indicators	Status of turn signals	Indicator status visualized on/off which blinks for every 2s when on, yellow icon if unavailable.
Light Status	Status of park, reverse, high beam, and low beam	Visualized using different icons, Error shows an exclamation mark on the icons
Break Light	Status of the break light	Indicated with red indicators behind a car's rear left and right side while on, black for off and yellow unavailable.
Distance Sensors	Obstacle detection from 0 to 2.5m for 10 sensors	The color-coded safety indicators around the car icon indicate proximity alerts according to the position of each distance sensor, using color and bar indications that will show the safety levels. For example, it uses red for danger and yellow for moderate. Red represents

		<p>danger: lights up one single red-colored bar when an obstacle is within 50 cm, which means that in such a case, there is a critical proximity warning. Yellow represents moderate: lights up two orange-colored bars if an object is detected from 50 cm to 150 cm and shows a medium safety level.</p> <p>Green-Safe: three green bars light up when the distance is between 150 cm and 250 cm-thus, a safe distance.</p> <p>Sensor Not Installed: Gray color scheme to indicate that the sensor has not been installed. Sensor Non-responsive: Orange color format to indicate if the sensor does not respond.</p>
LiDAR Sensor	0° to 360°, distance of 0 to 2.5m	<p>LiDAR data arrives in 10-degree increments and is packed into 12 segments of 30° to provide a simple, yet detailed, 2D representation of the surroundings.</p> <p>Partition Clustering: Since multiple values are collected in each 30-degree partition, only one closest distance metric was selected to take a representation for the whole segment.</p> <p>Red-Danger: When there is an obstacle within 50 cm from the vehicle, it shows one single bar red; this is an immediate proximity.</p> <p>Yellow-Moderate: Two bars of orange signify objects detected between 50 cm to 150 cm, showing a moderate distance.</p> <p>Green-Safe: This is when an object is detected between 150 cm to 250 cm, showing three green bars, indicating the safe zone.</p> <p>Sensor Not Installed: The grey colour shows where sensors are not installed.</p> <p>Non-Returning Sensor: If a sensor does not return any values, then the area highlights orange.</p>

<p>Switching between LiDAR and distance sensors</p>	<p>Lidar and Distance sensor based on the Speed</p>	<p>Dynamically selects and displays Distance Sensor or LiDAR data depending on vehicle speed and sensor availability.</p> <p>Distance Sensor Data: Displayed when the vehicle speed is less than 60 cm/s</p> <p>LiDAR Data: Shown when the speed is higher than 60 cm/s.</p> <p>Delay in Transition: Five seconds delay to ensure that the transition between the two displays is smooth enough. This means that crossing 60 cm/s requires waiting for 5 seconds for the system to toggle back to LiDAR data to avoid an abrupt change.</p> <p>Fallback to LiDAR: If none of the sensors indicating the distance are not working, then LiDAR data is used whatsoever the speed may be.</p> <p>Distance Sensors Fallback: When there is no LiDAR data, this fallback will display the distance sensor data.</p> <p>Lack of Speed Data: In case the speed data is missing, an error message will appear inside the Sensor Information display.</p> <p>Distance and LiDAR Not Available: If both the distance and the LiDAR sensor data are not available, then an error popup will pop up to warn the user.</p>
---	---	---

Thus, the above table summarizes Qt-based visualization requirements against each of the implemented features; ranging from data that will be visualized, up to edge-case outcomes. In the upcoming sections, how the application is programmed and is structured will be discussed.

4.3 Gateway Application on Raspberry Pi

Data from the Aurix microcontroller is captured by a CAN bus and sent to the Gateway Application running on the Raspberry Pi. It sends, in turn, the information to the CE Box via UDP. Further in this section, the architecture will be described, as well as the main components of the gateway application and code snippets to understand the logic behind gathering, processing, and sending data.

4.3.1 Overview of the Gateway Application

The Application Gateway on the Raspberry Pi is tailored to:

Interface your data from CAN bus through the PiCAN2 module, which works as an interface between CAN and a Raspberry Pi device. Filter this data further, thereby processing it in detail to find out the useful information about the vehicle.

Stream this information over a UDP network to the dashboard or visualization application running on another device, central control unit, or PC. The Hardware and Software Requirements include:

Table 4.4: Requirements for the Gateway application.

Hardware	Software
<ol style="list-style-type: none">1. Raspberry Pi 3B, with PiCAN2 CAN interface.2. Dashboard application running on a CE-Box, which will receive the data.	<ol style="list-style-type: none">1. Qt application on Raspberry Pi2. SocketCAN for CAN communication.3. UDP sockets for transmitting the collected data.

4.3.2 Gateway Application Structure and Code

The following sections describe the core functionalities and implementation of the gateway application, which consists of CAN data collection and UDP data transmission.

Project Configuration

The DataGateway.pro file is the **project configuration** file that manages the project's dependencies and build process. It defines the libraries and modules required to implement CAN and network functionalities, essential for data collection and transmission. Following are the qt modules that are include:

- **QT += core network**: This project uses the core module for general Qt functionalities and network for handling UDP transmission.
- **QT += serialbus**: Adds support for SocketCAN, a Linux-native CAN API, allowing the application to communicate with CAN hardware (PiCAN2).

CAN bus data collection:

The application initializes the SocketCAN interface, can0, for real CAN data collection from the PiCAN2 interface on Raspberry Pi. Below is a code snippet that shows how to set up and connect to the CAN device:

```
#include <QCanBus>
#include <QCanBusDevice>
```

```
#include <QCanBusFrame>
#include <QCoreApplication>
#include <QDebug>

// Initialize and connect to the CAN device on "can0"
QCanBusDevice* canDevice = QCanBus::instance()->createDevice("socketcan",
"can0");
```

- QCanBusDevice is an instance of the CAN device.
- Initialization of CAN interface - can0 in the code, which is particularly for PiCAN2, using QCanBus::instance()->createDevice("socketcan", "can0").
- If something goes wrong with the device connection, it logs an error message and exits.
- The above is setting up SocketCAN for the Raspberry Pi. SocketCAN is a CAN API based on Linux and is supported by the module PiCAN2.

Processing Inbound CAN Frames:

Here, when the CAN device is connected, it listens for the incoming frames, reads the data in them, and then processes it. That would further mean reading of Frame ID and payload of each received CAN frame.

```
#include <QUdpSocket>

QUdpSocket udpSocket;
QHostAddress udpHost("192.168.1.100"); // IP address of the receiver device
quint16 udpPort = 12345; // Port number on the receiver

QObject::connect(canDevice, &QCanBusDevice::framesReceived, [&]() {
    while (canDevice->framesAvailable()) {
        QCanBusFrame frame = canDevice->readFrame();

        QByteArray udpPayload;
        udpPayload.append(static_cast<char>(frame.frameId())); // Frame ID
        udpPayload.append(frame.payload()); // Frame payload

        udpSocket.writeDatagram(udpPayload, udpHost, udpPort);
        qDebug() << "Forwarded CAN frame via UDP:" << udpPayload.toHex();
    }
});
```

- CAN Frame ID and Payload: Appending the CAN frame ID as the first byte, followed by the payload data.
- UDP Transmission: The function udpSocket.writeDatagram() sends the CAN data via UDP to a specified host and port.

Formatting CAN Data for Transmission:

The byteArrayToHexWithSpaces function, which formats udpPayload into a readable hex string, assists in debugging and visual verification of the data structure.

```

QString byteArrayToHexWithSpaces(const QByteArray& byteArray) {
    QString hexString = byteArray.toHex();
    QString spacedHexString;
    for (int i = 0; i < hexString.length(); i += 2) {
        spacedHexString += "0x" + hexString.mid(i, 2).toUpper();
        if (i + 2 < hexString.length()) {
            spacedHexString += " ";
        }
    }
    return spacedHexString;
}

```

Setting up UDP Transmission:

Instantiates a `QUdpSocket` to handle UDP transmission. It sets the destination IP and the port so that it can connect seamlessly with the destination device, in this case, the CE Box. Sends the `udpPayload` after the processing of each frame to the CE Box for visualization.

```

QUdpSocket udpSocket;
QHostAddress udpHost = QHostAddress::Any;
quint16 udpPort = 12345;

```

The Gateway Application is an intermediary that takes on the task of translating the raw CAN data into a format suitable for network transmission. `QCanBus` is used within the application to handle the CAN messages, and `QUdpSocket` is used for transferring the data in order that the Raspberry Pi transmits data effectively in real time to the CE Box. Each frame will be read, processed, and transmitted in order with very low latency such that the visualization experience on the dashboard application will be agile and responsive.

4.4 Visualization Application on the CE-Box

The visualization application on **the** Central Control Unit (CE Box) provides real-time feedback about the car's status by receiving CAN data over UDP, processing it, and rendering it on a user-friendly dashboard. This application is designed with Qt/QML, which offers powerful tools for graphical user interfaces.

Project Configuration

The `Dashboard.pro` file, apart from the other source files in this folder, is a configuration file for the project. As mentioned, it's very important because it enumerates several dependencies, resources, and files needed to compile the application. It defines key components like:

`QT += quick core serialbus`: This line specifies that the project uses the Qt Quick module for UI, core for basic Qt functionalities, and serialbus for CAN communication.

Sources: Lists C++ source files (main.cpp, UdpReceiver.cpp), which handle backend operations like data reception and processing.

Resources: Specifies the QML resources (qml.qrc) that are used in the application for UI components.

The Dashboard.pro file configures the project's build process, ensuring that all necessary modules and resources are included for successful compilation and deployment.

Main Entry Point

main.cpp initializes the Qt application, sets up the UDP data receiver, and loads the main QML file, containing the user interface, namely main.qml. The key functionalities include:

Application Setup: Creates QApplication and instantiates the UdpReceiver type so it's available from QML for data reception.

Connection between Engine and Object: QQmlApplicationEngine connects to load the main.qml file and handle the application errors.

This file effectively bridges the back-end C++ components with the QML UI and makes the UdpReceiver class directly accessible inside QML for real-time updates.

```
QGuiApplication app(argc, argv);
QQmlApplicationEngine engine;
qmlRegisterType<UdpReceiver>("Dashboard.UR", 1, 0, "UdpReceiver");
engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
```

Data Receiver: UdpReceiver.cpp and UdpReceiver.h

The UDP receiver class receives the CAN data in UDP and interprets it. Every can frame is parsed by means of filtering useful information to send to the dashboard:

UdpReceiver.cpp:

UDP socket set up: Creates a QUdpSocket and binds it to port, and starts listening for incoming packets.

Data Processing: This involves parsing CAN frames to provide values at the application level for various indications, such as speed, engine status, and sensor data. Handle the timeouts accordingly. If in that time, data is not returned, mark some fields erroneous so that at least the accuracy and responsiveness are maintained.

```

connect(udpSocket, &QUdpSocket::readyRead, this,
&UdpReceiver::processPendingDatagrams);

void UdpReceiver::processPendingDatagrams() {
    QByteArray datagram;
    udpSocket->readDatagram(datagram.data(), datagram.size());
    processCanData(datagram);
}

```

UdpReceiver.h:

Declares functions returning the parsed data. For instance, `get_Speed()`, `get_Engine()` etc., then exposes them to QML, which in turn can be used to update UI elements. It acts like a data link between the CAN network and the visualization dashboard, through which data from the car is continuously received and processed.

Main QML File

Main.qml is the major QML file, comprising the layout and arrangement of UI elements on the dashboard. It would comprise instances of custom QML components, such as `G_ArcGauge`, `G_Image`, and `G_Icon`, that would come in handy for the visualization of various car status aspects.

Data Binding: The binding of data coming from `UdpReceiver` functions to update UI elements in runtime.

Indicators and Gauges Examples of custom views to display speed, battery status etc.

Timers and Update Logic: In case of encountering a `Timer` element, it constantly updates the data by calling functions from `UdpReceiver`.

```

UdpReceiver {
    id: urObj
    function updateSpeed() {
        arcGauge.requestPaint(urObj.get_Speed());
    }
}
Timer {
    interval: 500
    repeat: true
    onTriggered: urObj.updateSpeed()
}

```

Custom QML Components

These custom QML components improve the reusability and readability of code. Each component encapsulates specific functionalities of the dashboard.

G_ArcGauge.qml:

`G_ArcGauge` is a custom gauge component and is used to display circular indicators like speed or battery status. This custom component uses the `Canvas` element to draw a partial arc, which represents value proportionally.

- **Data Driven Drawing:** the requestPaint function is invoked with parameters, such as value and unit. Depending on the received parameter, the arc length will change. Labeling - Minimum and maximum values displayed to improve readability. These custom QML components enhance reusability and readability of the code. Each component encapsulates specific dashboard functionalities:

```
Canvas {
    onPaint: {
        var ctx = getContext("2d");
        ctx.arc(width / 2, height / 2, radius, startAngle, endAngle,
false);
        ctx.stroke();
    }
}
```

G_Icon.qml:

G_Icon displays an icon with a status indicator. It uses a base image and can toggle to an error icon based on specific conditions, such as errors received in CAN messages.

```
Image {
    source: _isErr ? "error_icon.png" : _imgPath
}
```

G_Image.qml and G_Image2.qml:

They provide general-purpose image components for various kinds of indicators in this dashboard. For example, images may appear according to some dynamically set path properties in main.qml.

HorizontalGauge.qml:

HorizontalGauge is a horizontal gauge component that can be used to display battery or fuel levels in a linear fashion. This puts a different visualization style compared to the arc gauge.

4.4.1 Queued Connection and Single-Slot Mechanism

Queued Connection:

In the Dashboard Project, the Queued Connections are mainly used for the interaction between the UdpReceiver object and QML. A Queued Connection delays the execution of a slot-a function connected to a signal-until the main event loop is ready. This is useful when there is heavy data processing involved or running on resource-constrained hardware, such as embedded systems, for example, Raspberry Pi, in that it keeps other tasks running uninterrupted.

Queued Connection: Used when a signal emission is done on C++ backend, for instance; when data arrives over UDP in UdpReceiver and the slot to be invoked is implemented in QML to update the UI elements.

Qt::QueuedConnection ensures asynchronous evaluation of the objectCreated signal within the main event loop. This is crucial to guarantee good QML responsiveness by not keeping the UI blocked.

```
QObject::connect(&engine, &QmlApplicationEngine::objectCreated, &app,
[url](QObject* obj, const QUrl& objUrl) {
    if (!obj && url == objUrl) QCoreApplication::exit(-1);
}, Qt::QueuedConnection);
```

Single-Slot Mechanism:

Each signal gets connected to one slot. In the case of the Dashboard Project, it means that in handling the data from UDP connections, a signal framesReceived in the process of receiving CAN data is hooked directly to a single slot processing the frame data and forwarding it to QML. Single slots are used in order not to create unnecessary complexity, where every signal-for example, the arrival of new CAN data-will result in a single update operation in the UI.

4.4.2 Class Diagram for the Dashboard application

Application receives CAN data through class UdpReceiver; it then processes data and extracts values to be shown. Values are refreshed periodically in QML UI using the binding property of custom components. For example, G_ArcGuage, G_Image. Each of the custom components in QML files represents some part of car status. For example, speed, battery, indicators. QML elements bound on the properties of UdpReceiver get updated in real time, while keeping the status of a car on the dashboard correctly.

Amazingly, the CE Box is an interactive Visualization Application used for real-time status monitoring of a vehicle. This application is modular, extendable, and responsive in terms of car diagnostics visualization with Qt/QML for UI and UDP for sending data. This application's structure, in its way or by using built-in QML components, allows rendering CAN data in an application in a pretty flexible and precise way. Below is the class diagram for the Dashboard application.

Dashboard Application Class Diagram

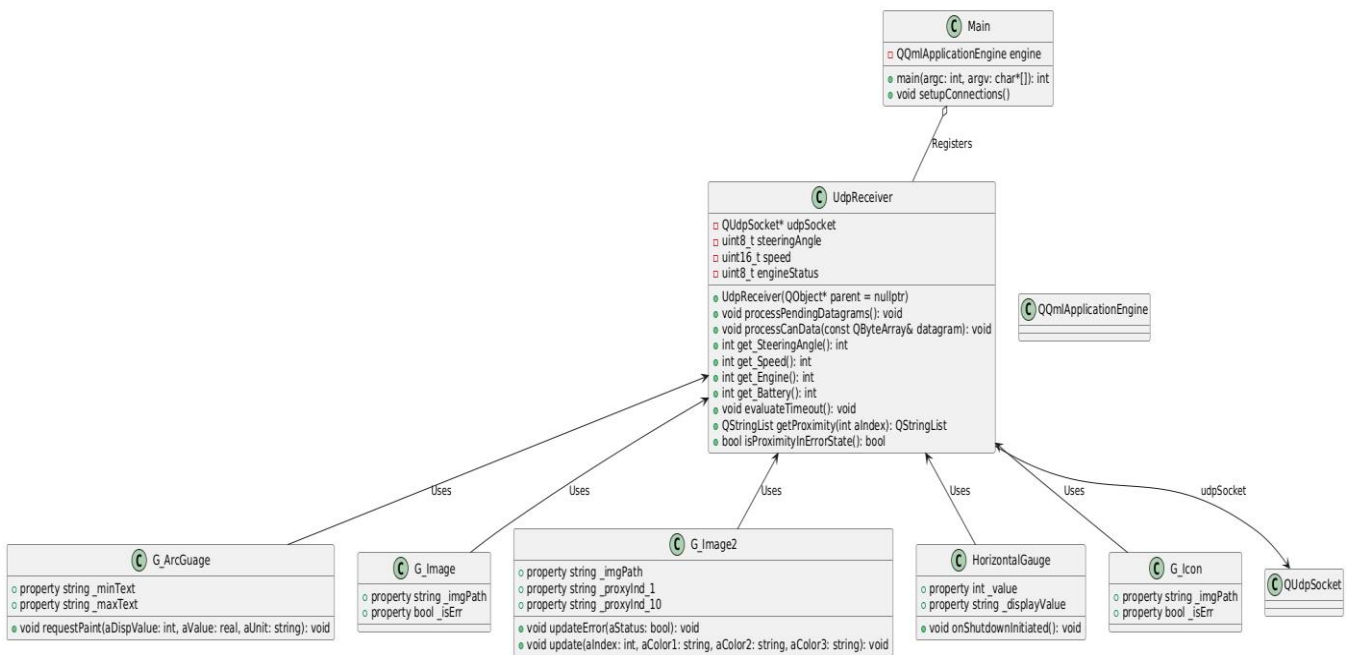


Figure 4.1: Class Diagram of Dashboard Application.

4.5 Network Configuration and Error Handling

The system network protocols, data flow, and configuration settings are described here. This is particularly relevant for the dashboard application, since the dashboard application depends on UDP for receiving CAN data from the Gateway application. The details of the choice of protocols and port and IP settings come very relevant.

Network Protocols:

It sends information to the gateway running at the Raspberry Pi via a UDP network. UDP is used for its low-latency nature, hence allowing real-time updates on the dashboard visualization on the vehicle's dashboard display. However, UDP is connectionless and thus cannot guarantee that a message will be delivered. Therefore, UDP suits scenarios where minor packet losses are inconsequential. It includes:

CAN Data Gathering: Raspberry Pi Gateway will read data from in-car CAN via SocketCAN on can0. **UDP Transmission:** Preprocessed data at the Gateway is transmitted as UDP packets to the dashboard application. **Binding of Port:** The Raspberry Pi binds itself to a UDP port and sends packets to an IP address and port in the CE Box where the dashboard app is listening.

Configuration Settings: These different ports are utilized along with IP addresses to configure both the gateway and dashboard applications. For instance, Qt uses

QHostAddress for dealing with the destination IP, while QUdpSocket handles sending and receiving of data.

Pre-Deployment Network Validations:

- IP Address Setup: Raspberry Pi and the CE Box should be in the same subnet for UDP to directly communicate between the two.
- Initializing CAN Network: CAN initialization on Pi should be done through the loading of kernel modules for 'can', setting of 'can0' using the command 'sudo ip link', and defining an appropriate bitrate; for example, 500000.
- Testing Tools: Linux utilities such as can-utils enable the user to simulate and monitor CAN messages. Network utilities like netcat or Wireshark should also be able to assist in the validation of UDP packet flow.

This would involve creating a UDP socket and binding it to either localhost or an IP on a port in preparation for sending data to the dashboard.

```
QUdpSocket udpSocket;  
QHostAddress udpHost("192.168.1.100"); // CE Box IP Address  
quint16 udpPort = 12345;  
  
udpSocket.writeDatagram(udpPayload, udpHost, udpPort);
```

Error Handling and Data Validation

This section should explain how the system ensures that the information is not corrupted and is valid, since it is very valuable for automotive applications in real-time.

Mechanisms of Error Handling

1. CAN Data Timeouts: Checks on times-out for messages on the CAN data are performed by the class UdpReceiver. If something doesn't show up after some time, then the data is considered stale or wrong. Due to this, any old or wrong information is not reflected on the dashboard to the user.
2. Abnormal Termination: The gateway application uses signal handling so that graceful closure can be allowed. This makes resources like CAN and UDP connections shut down correctly in case sudden exits avoid data corruption or memory leakage.
3. Control of the proximity sensor state: The proximity data needs to check the errors to get into the valid or not status. If the proximity sensor seems to be in an error status, the fallback visuals or alerts will be enabled for the application to notify the user.

4. QML UI Validation: Many UI elements would check error conditions and time-outs. For example, fallback values or icons would be shown if speed or the battery indicators are not available. This could be implemented in QML using conditional statements that simply turn on error icons.

Data Validations

1. CAN Frame Validations: Every incoming CAN frame to the gateway will be validated for the format and length of the data as expected before processing.
2. Range Checks: The critical indicators, such as speed and battery level, range validate to avoid incorrect presentations or out-of-bound values.
3. One Touch Data and Redundant Checks: Application shows the fallback indicators in case some data gets flagged in error, so that it will not show incorrect values because of that and will maintain smooth user experience.

Error handling and data validation provide the system with reliability, safety, and user-friendliness. Each module-from the low-level gathering of CAN data to rendering QML-maintains checks for discrepancy in data or signal failure. These methods ensure that resilience within the application can provide runtime dependability on the visualization of vehicle status despite inconsistency at the network or sensor level.

4.6 Sequence Diagram

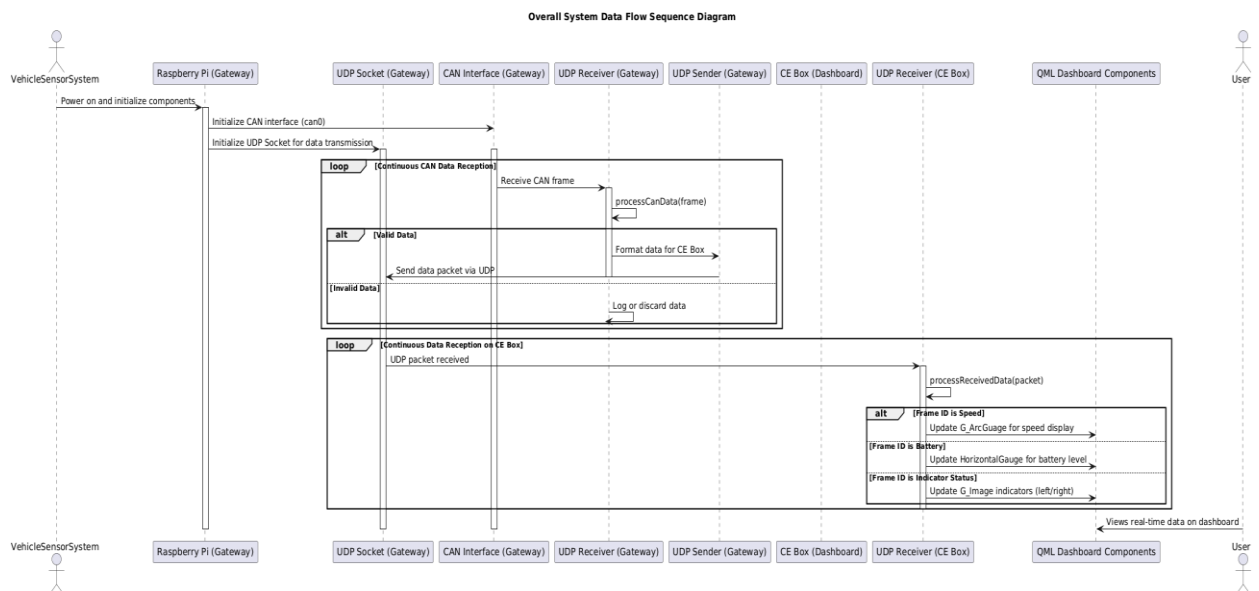


Figure 4.2: Sequence diagram for the application.

Diagram Elements Explanation

VehicleSensorSystem: This may be thought of as a sensor system on a vehicle which would be generating the sensor data to be sent across CAN. Example parameters may include speed, battery level, and indicator status.

Gateway on Raspberry Pi:

Raspberry Pi Initialization: The CAN and UDP components are initialized to handle the data from the vehicle.

Data Collection: Receive frames from the sensor system running on CAN, process this data, and send required data across UDP to the CE Box

Error Handling: Data not recognized or invalid would either be dropped or logged to file.

CE Box-Dashboard Application:

Data Reception: This involves receiving UDP packets from the Raspberry Pi. UdpReceiver_CE running on CE Box processes these packets.

Component Updates: According to data type as notified by CAN frame IDs, update QML UI components like G_ArcGauge for speed, HorizontalGauge for battery, and G_Image for indicator lights.

User Interaction:

Real-time data will be viewed by the user on a dashboard, which keeps updating itself with continuous and changing incoming data from Gateway into the CE Box.

Implementation of Key Components • UDP Socket: UdpSocket_RPi and UdpReceiver_CE-Facilitate network communication between Gateway at Raspberry Pi and CE Box to deliver transformed packets of information.

QML Components in Dashboard: Following QML elements -G_ArcGauge, HorizontalGauge, G_Image are updated dynamically depending on the data input provided to give runtime visualization to the user.

4.7 Dashboard UI Design and Layout

Canva was used to finalize the visual design of the Dashboard UI and its prototype before actual development. The UI is designed in such a way that all components are put on one single screen for instant visibility of the user on the essentials of vehicle data. It was well thought out how each element will be placed intuitively and immediately interpretable by both numeric values and visual indicators.


Key design choices: these include sliders for battery level and engine power. Sliders were adopted for continuous value representation to provide smooth real-time feedback of the power levels. The speedometer makes use of half-circle arcs. An arc displays the speed graphically by sliding, while numeric values are shown below the arc for clarity.

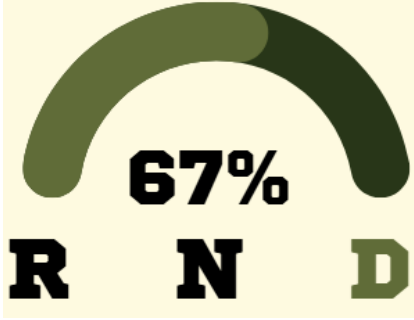










Icons - Tell Tales: Included are custom icons for a variety of status indicators such as headlights, blinkers, and other vehicle lights. The icons light up in color depending on availability and turn gray with an overlaid exclamation mark when the indicators are unavailable.

Central Car Representation: Centrally located is a car icon that shows brake lights to indicate proximity by using feedback from the distance and LiDAR sensors. The proximity information is visualized by color-coded lines around the car, in red, yellow, and green, depending on the obstacles detected.

Below is a summary of each of the visual components in tabular form to make clear how each item was conceptualized for clarity in the representation of specific vehicle data. Based on this design rationale, the Dashboard UI could then present a consistent, informative, and user-friendly display to meet the visualization requirements of the TUCminiCar system.

Table 4.5: UI Components Overview.

Component	Description	UI Representation
Speedometer	Displays the speed range from 0 to 120 cm/s	

Power meter	Displays the engine power in percentage, with D when forward power from 0 to 100; R being reverse with negative 0-100 percent and N being neutral.	
Battery	Displays battery voltage from 0 to 100 in percentage	
Steering wheel	Displays the directions of the steering- right, left and center	
Brake lights	Indication on the rear right and left of the car. Red- on; White- Off and orange- Unavailable	
High beam	On and off telltale shown	
Low beam	On and off telltale shown	
Park Lights	On and off telltale shown	
Right Indicator	On and off telltale shown with blinking	
Left Indicator	On and off telltale shown with blinking	
Error Icon	Shown on the tell tales when the values received are out of range or unavailable	
Distance and Lidar sensor	Color coding: - Red: < 50 cm (danger- 1 bar) - Yellow: 50-150 cm (moderate- 2 bars) - Green: >150 cm (safe, not visualized-3 bar) - Orange: no signal - Grey: not available	

The dashboard user interface design, which is visual and user-friendly for each key element that represents data of the vehicle. This interface displays the overall critical information together to achieve ease and speed in observing real-time information. It also contains the top-mounted battery status along with a respective low-battery warning indicator-on in case the output of it goes below the threshold predefined by a value. On the left-hand side of the screen, an arc-shaped gauge acts as the speedometer, showing the current speed of the vehicle in a graphical manner. Symmetrical to this, on the right-hand side, another arc gauge displays the engine power or battery level, clearly and dynamically representing the performance metrics of the vehicle. These gauges are designed to be both visually engaging and easy to interpret briefly.

In the center of the dashboard, a car icon serves as the center for proximity information from both the distance and LiDAR sensors. Color-coded lines radiating around the car icon will indicate the proximity of obstacles: red for danger, yellow for caution, green for safety, and orange means unavailable sensor data. Thus, this provides the user with an easy way to assess surroundings. The car icon also shows the other indicators, such as the extra break lights. When this button is pressed, after an instant, it illuminates red. Below the power gauge, there is the gear status indicating the position of the vehicle when on neutral, reverse, or in drive mode.

For further convenience, supplemental icons for parking status, high and low beams for the headlights, and turn signals line up across the bottom of the screen. The icons are for quick reference and change dynamically to reflect the vehicle's current state. All these put together combine into a well-structured intuitive interface that supports effective decision-making and overall good user experience.

5 Testing and Evaluation

This chapter will discuss on how the implementation was tested along with the discussion of the reuts. Since it i challenging to cover the corners of the real world scenarios, a simulation tool with gui was developed to test the qt dashboard application which is already discussed in the previous sections.

5.1 Testing Environment Setup

5.1.1 Hardware setup

The following setup involves the deployment and interconnection of three major parts: the Dashboard on the CE Box, DataSimulator on a Linux PC, and DataGateway in order to provide wireless transmission of CAN data to Wi-Fi. It is thus capable of simulating on the same platform the correct visualization of the data on the CE Box based on the already simulated CAN signals coming from the Linux PC. This allows a good exposure of real-life scenarios for testing.

1. Dashboard: The Dashboard is a Qt-based application developed to show real-time data transmitted over Wi-Fi. It is deployed on the CE Box, which is connected to a touchscreen display for interactive monitoring. It listens for UDP packets transmitted by the DataGateway application and parses the information to update the vehicle metrics on the screen.

Deployment: The Dashboard application should be compiled and executed on the Raspberry Pi in the CE Box to interact in real-time, visualizing the data of the vehicle.

2. DataSimulator: It is a GUI application designed to simulate most of the CAN messages that would normally come from sensors and actuators from the MiniCar.

This GUI will allow a user to toggle settings on and off, slide sliders, and simulate a myriad of data conditions. DataSimulator sends CAN messages over the virtual CAN (vcan) interface to be processed by the DataGateway onto be forwarded for display by the Dashboard.

Deployment: DataSimulator is compiled and executed on a Linux PC, preferably Ubuntu where it will serve as a primary source for simulated CAN data in testing.

3. DataGateway: It is a console application that sends simulated CAN data from the PC over vcan to the Dashboard on the CE Box via Wi-Fi. It parses CAN messages output by the DataSimulator and converts them into UDP packets to send across to the CE Box so that the Dashboard displays real values in real time.

Deployment: The DataGateway application will be developed and executed on the Linux PC together with the DataSimulator; it captures the CAN data and takes care of the transmission.

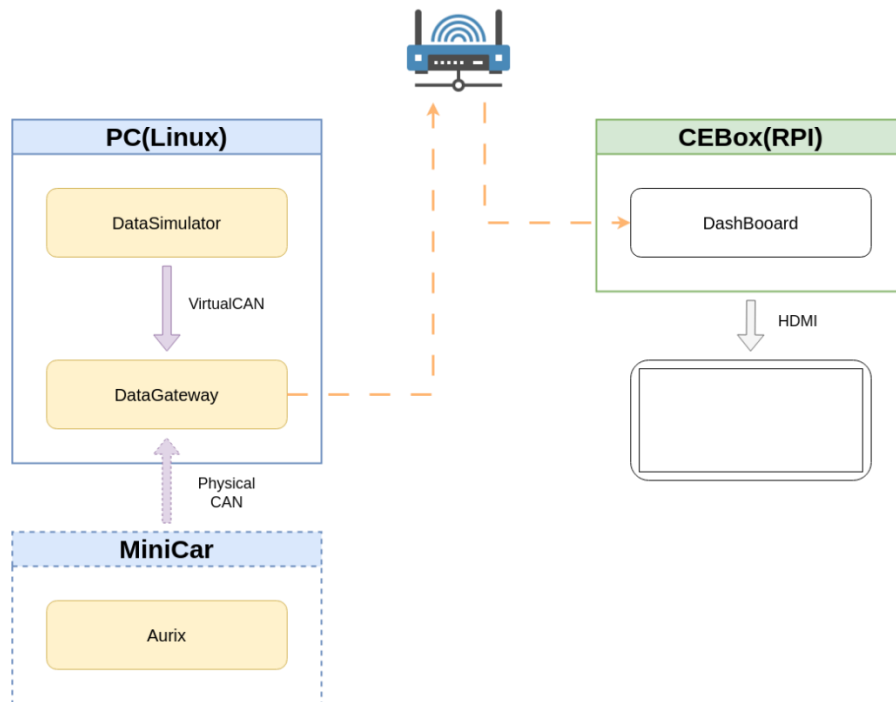


Figure 5.1: Setup for testing.

5.1.2 Data Simulation Application

This section describes the design and functionality of the Data Simulation Application using the Qt framework. This application provides a UI with which the user can interactively simulate various conditions that will be encountered in a car system. The buttons toggle and sliders adjust in such a way that dynamic creation and sending of CAN messages can be done for testing the visualization features of the CE Box dashboard application. The Data Simulation Application is one end-to-end application intended to simulate real automotive scenarios without using physical sensors or any other hardware input for the same. It's meant to confirm the Dashboard's ability to update itself correctly upon changes in the state of the car using controlled and customizable CAN messages.

QML UI Elements:

Sliders: These sliders represent different parameters which might need to be tuned - examples are speed, steering angle, and even battery level. Each slider element has minimum, maximum, initial, and step values that allow the user to adjust values smoothly. A Check Box is added to each slider. In case any of the Check Boxes is changed, enabling/disabling of the data input for the particular parameter is carried out.

Indicators: This is for the simulation of status indicators such as turn signals or high/low beam statuses. The said indications, when switched on/off, send specific CAN signals to simulate the activation/deactivation of those features, making the dashboard reflect such an instance as it would in real life.

Backend C++ class: The respective backend C++ code will capture the values provided from QML UI elements and will pack them into CAN messages. DataSimulator class provides messages that are in the pattern/structure expected by the CE Box Dashboard application. The messages are formatted according to the CAN protocols along with necessary identifiers and data fields.

CAN interface network communication: The application is configured to send/receive information from/ through a virtual CAN interface, for example -vcan0- or in case connected with hardware by the can0 network interface. All the simulated messages are forwarded outside in real time to test and visualize them in the CE Box.

Application Flow

User Input: User modifies some vehicle parameters and indicators using sliders and check-boxes.

Generation of CAN Message: The changed values from every slider or toggle component are read by the backend DataSimulator class, creating a corresponding CAN message that represents the simulated condition.

Message Transfer: These kinds of messages are transferred to the CAN network that is connected to the running dashboard application on the CE Box. Real-time transmission allows it to have instant visualization on the dashboard interface.

Test and Validation

The Data Simulation Application will provide a strong test environment for the dashboard application in several ways, including enabling functional and performance testing of the application. In this regard, the simulation of various driving conditions and states of the vehicle will enable testing of the dashboard response to CAN messages for correctness. This could include performance tests, such as latency and responsiveness in terms of the updates on the dashboard.

Data Visualization Validation: Once the amount of simulated CAN messages is seen by the reaction of the CE Box, one may well reassure that each message was correctly interpreted and visualized on the dashboard.

The tool thereby serves as a comprehensive test interface for realizations of detailed and repeatable test scenarios and helps in the validation of functionality of the CE Box dashboard application.

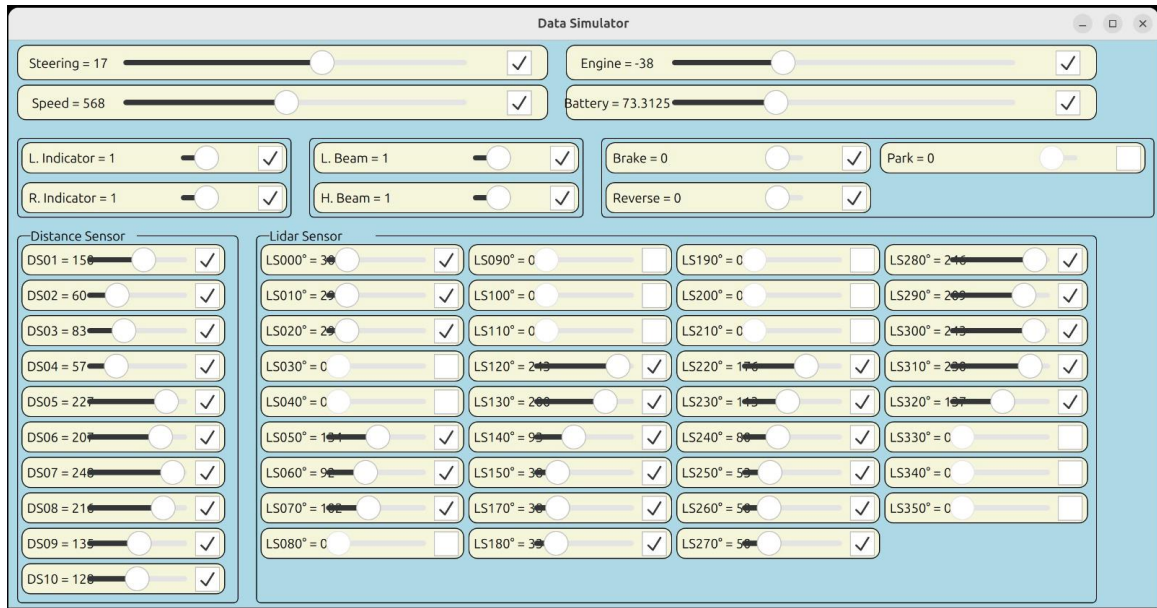


Figure 5.2: UI of the Data Simulator.

5.2 Functional Testing of the Qt Application

Functional Testing Functional testing shall be performed to ensure that Qt behaves as expected in different situations: It has to display the vehicle data properly and handle both normal and edge cases. This will involve all the different parts of the dashboard: Each gauge and each indicator, and each central car icon is operating correctly and providing helpful feedback to the user. The main focus of functional testing will be on:

- **Data Representation Accuracy:** Ensuring that any of the UI elements-for example, a speedometer or battery level-will be showing the right data provided through CAN messages.
- **Edge Case Response:** It checks the behavior of the dashboard at the boundary conditions, which includes minimum and maximum values, missing data, and unexpected data inputs.
- **Responsiveness of User Interface:** The UI should refresh itself in real time with the least latency, therefore providing an experience to the user that is frictionless.
- **Error Handling and Alerts:** There are warnings or other error messages that the system needs to give out based on the out-of-range, not available, or malformed data.

In further sections different testing process will be discussed which has been tested according and has passed the test cases.

5.2.1 Basic Functional Tests

In this section different test cases scenarios are discussed which not only been tested but also all the test cases are passing.

Sanity Check: It is performed to observe if the major components are functional properly which would not affect a major or whole part of the application.

Summary	Test Steps	Expected / Observed Outcome
System should be up and running will all the UI components visible and functional	<ol style="list-style-type: none"> 1. Power on the Display and connect to the TUCMiniCar 2. Make sure the can device is setup 	<ol style="list-style-type: none"> 1. Application running: The CE Box should boot up successfully and launch the dashboard application without any loading errors or crashes. 2. UI viewability: All UI elements should be clearly visible, centered and undistorted on the display of a speed gauge, battery indicator and proximity sensors, to name some. 3. Component functionality: The UI components shall react to stimulated CAN messages in a manner that represents the update of data in real-time and with no delay.

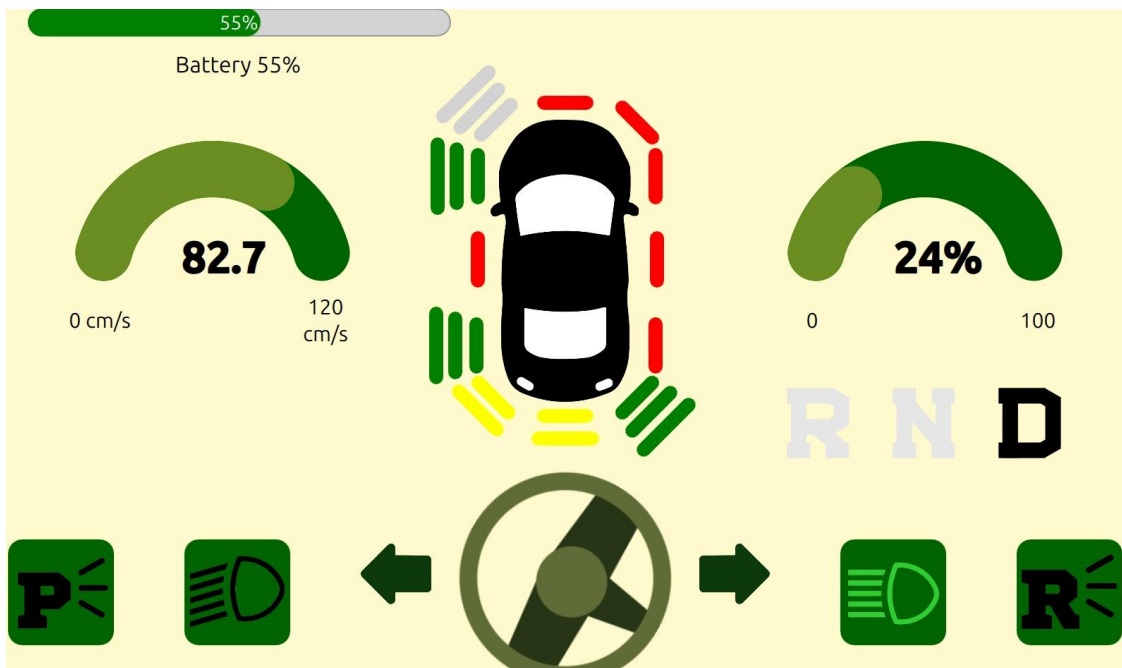


Figure 5.3: All elements visible and functional.

Working of Lidar and Distance sensor

Summary	Test Steps	Expected Outcome
<p>Distance sensor data visible when the speed is below 60cm/s</p>	<ol style="list-style-type: none"> 1. Use the DataSimulator to set the vehicle speed below 60cm/s. 2. Send CAN messages to simulate distance sensor and lidar sensor data at different proximity levels (e.g., 50 cm, 100 cm, 200 cm). 3. Observe the dashboard to check if distance sensor data is displayed around the car icon. 	<ol style="list-style-type: none"> 1. Distance sensor data should be visible on the dashboard, showing proximity indicators around the car. 2. Data should be color-coded according to the distance (e.g., red for <50 cm, yellow for 50-150 cm, green for >150 cm). 3. Previously if the lidar data was displayed, then the switch should happen within 5s of time interval
<p>Lidar sensor data is seen when the speed is above 60cm/s</p>	<ol style="list-style-type: none"> 1. Use the DataSimulator to set the vehicle speed above 60 cm/s. 2. Send CAN messages to simulate LiDAR sensor and distance sensor data with various proximity readings. 3. Observe the dashboard for changes to LiDAR visualization around the car icon. 	<ol style="list-style-type: none"> 1. LiDAR data should replace distance sensor data on the dashboard when speed exceeds 60cm/s. 2. Proximity indicators should change based on LiDAR readings, using the same color-coding scheme. 3. Previously if the distance sensor data was displayed, then the switch should happen within 5s of time interval

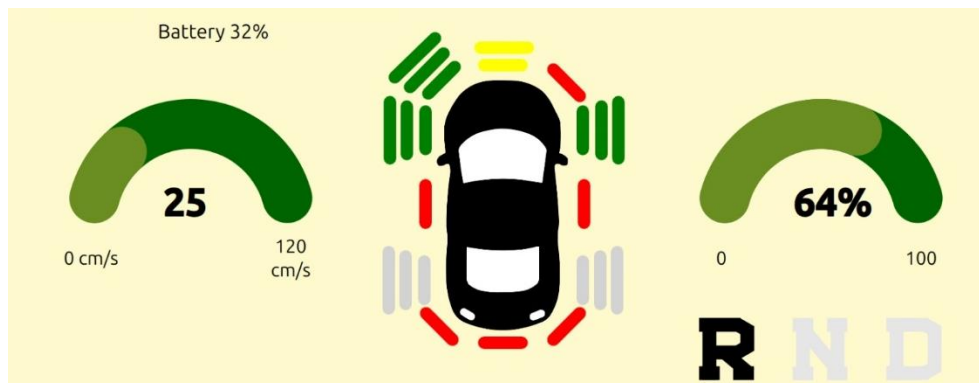


Figure 5.4: Distance sensor data when speed <60.



Figure 5.5: Lidar data displayed when speed >60.

Battery, brake lights and power meter

Summary	Test Steps	Expected Outcome
<p>Battery voltage should be represented in percentage; warning text and color, orange, if below 15%</p>	<ol style="list-style-type: none"> In DataSimulator, the battery voltage data at different values is set, for example 50%, 20%, 10%. In the dashboard, observe the battery percent slider. The values should change. Be very attentive to the test for a value below 15% that returns the warning. Provide unavailable or error message simulation. 	<ol style="list-style-type: none"> Battery level to show the proper percentage on the dashboard. If the battery level reaches below 20%, a warning message shall appear and slider will be orange When below 5%, battery critically low pop up with will be blinking with slider turned to red.

<p>Power percentage in forward or reverse direction Display showing highlighting of N/R/D, Neutral/Reverse/Drive</p>	<p>1. Using DataSimulator change the pedal percentage forward and reverse. Messages 2. Observe the dashboard and confirm that the pedal percentage and gear status [N/R/D] are provided and highlighted correctly</p>	<p>1. Power or Pedal Percentage should reflect the correct percentage according to input for example, 30% forward or 40% reverse. Depending on the pedal mode, the corresponding gear status will be highlighted, such as N, R, and D.</p>
<p>The brake lights appear on the car icon.</p>	<p>1. Using DataSimulator, toggle the brake light signal from "on" to "off". 2. Make the brake light signal simulate "unavailable" or "error". 3. Observe the Brake light indicators on the car icon in the Dashboard</p>	<p>1, (rear-left and rear-right). ':' In case of 'on' brake light signal, the brake lights on the car icon rear-left and rear-right shall glow red. In case of the 'off' brake light signal, the Brake light shall be displayed in Black. 2. If the brake light signal is not available or faulty, the brake lights should appear in orange to indicate visibly.</p>

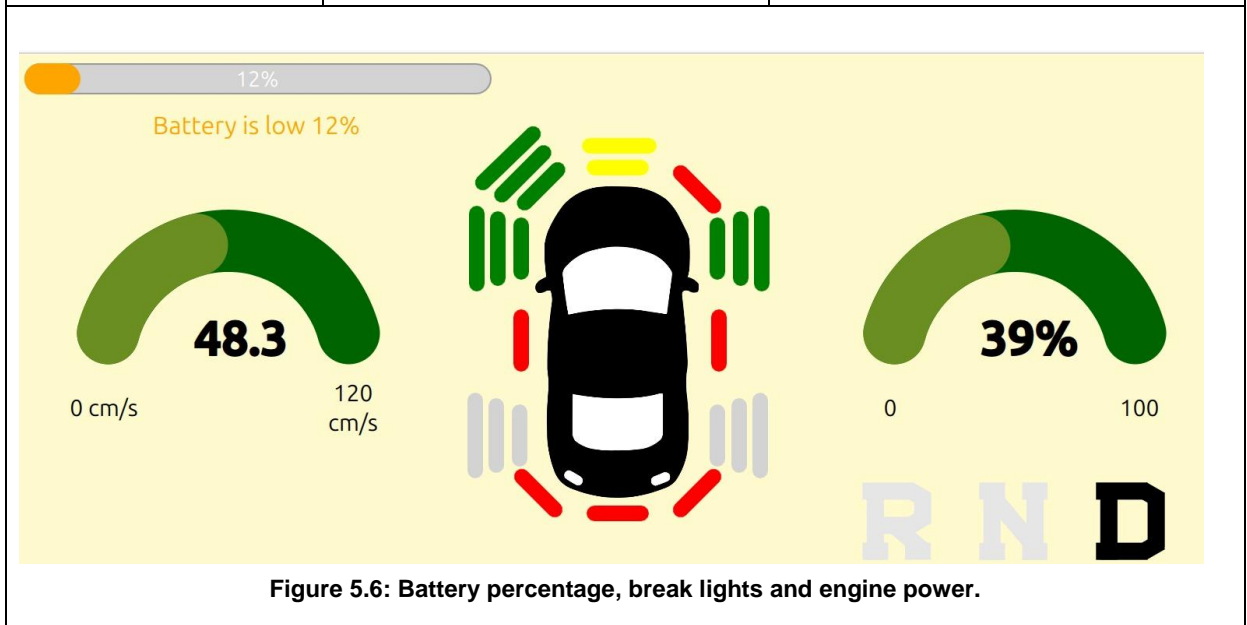


Figure 5.6: Battery percentage, break lights and engine power.

Steering wheel, Turn Indicators, and lights:

Summary	Test Steps	Expected Outcome
Right / left rotation of steering wheel icon	<ol style="list-style-type: none"> Using DataSimulator, update the status of the steering to left, neutral, right. Update the status to unavailable or a fault. Observe the steering indicator icon on the dashboard. 	<ol style="list-style-type: none"> The steering wheel icon reflects the status left, neutral, or right based on the input. When the status is unavailable, a warning symbol, such as an orange exclamation mark, overlaid on the icon.
Show turn indicators (left/right) with blinking effect.	<ol style="list-style-type: none"> Use DataSimulator to enable/disable left and right turn indicators. Indicator status unavailable. Observe the UI 	<ol style="list-style-type: none"> When the left/right indicator is on, icon blinks on for 400ms and off for 400ms When the indicator is off, icon is OFF When data is unavailable, icon goes yellow to indicate status unavailable.
Light status-Park, Reverse, High Beam, Low Beam of the vehicle	<ol style="list-style-type: none"> Using Data Simulator, switch ON/OFF the park, reverse, high beam and low beam lights. Mark the status of each light as unavailable. Check icon status of each light on the dashboard 	<ol style="list-style-type: none"> Corresponding icon in active color lights up when any light is on. The icon should be black or default color for an inactive state when any of these lights are off. When any of the data from these lights is unavailable, it colors orange. <p>An icon pops up with an exclamation mark whenever there is a problem.</p>



Figure 5.7: Steering and the lights.

5.2.2 Error and Alert Display Tests

Summary	Test Steps	Expected Outcome
When Speed and Engine Power are not available	<ol style="list-style-type: none"> 1. In the DataSimulator, make both Speed and Engine Power data unavailable. 2. See the dashboard display for speed and engine power gauges. 	<ol style="list-style-type: none"> 1. Instead of values in the dashboard, it should display -- for Engine Power and 0 for speed. 2. All other UI parts shall not be affected, and the rest of the interface is working accordingly.
When battery is 0%	<ol style="list-style-type: none"> 1. Using the DataSimulator, set the battery level to 0%. 2. Observe dashboard for messages that should be shown on screen and changes to the screen. 3. After the warning appears, wait 5 seconds to confirm behavior of shutdown. 	<ol style="list-style-type: none"> 1. When the battery reaches 0%, the entire screen should display a critical low battery warning on a completely black background. 2. Dashboard UI should turn off automatically after 5 seconds.
When any one of the sensor data (lidar or distance sensor) or speed is not available	<ol style="list-style-type: none"> 1. Simulate unavailability by stopping the data feed for either the LiDAR or distance sensor or speed (e.g., turn off messages) 2. Observe how the dashboard handles missing data from either sensor. 	<ol style="list-style-type: none"> 1. If one sensor is unavailable, the dashboard should continue displaying data from the other sensor. 2. If both the sensors and the or the speed data is unavailable, then error message on the car icon saying distance and lidar has error pop up is soon.

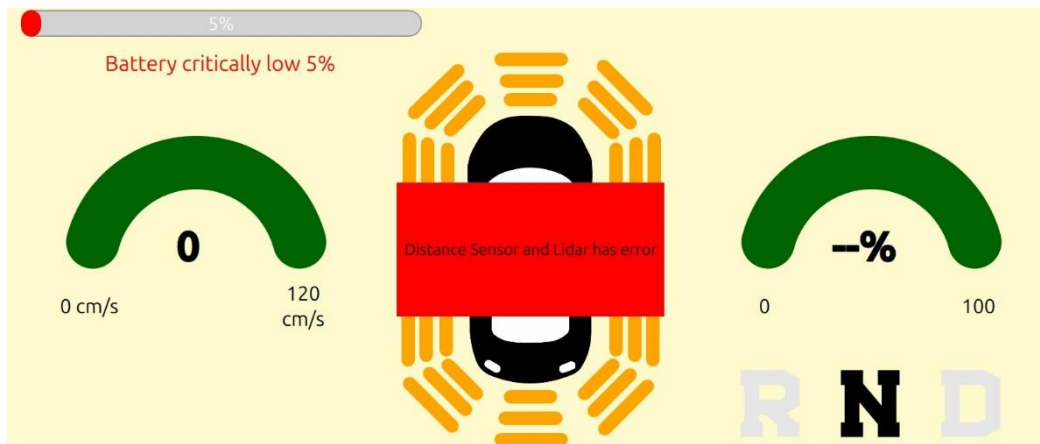


Figure 5.8: Sensors unavailability.



Battery Critically low, System will power off

Figure 5.9: When battery 0%.

5.3 Testing with Tiny-Can

The initial validation and refinement of the Gateway Application were done in a lab environment using the Tiny CAN [47] module. The application deployed on the Raspberry Pi, connected with Tiny CAN, allowed for the precise testing of handling CAN messages, accuracy of transmission, and dashboard updates to ensure a smooth flow of data, reliably delivered from the virtual CAN bus to the CE Box.

Tiny CAN is a professional, compact, lightweight USB-to-CAN adapter widely used for test purposes, simulation, and development of applications based on the CAN principle. Due to its low weight, it is compatible with USB, thus easily connectable to systems like a PC or Raspberry Pi. Tiny CAN allows the usage of standard CAN protocols, hence providing an easy way to communicate and test in environments where there is a necessity to create, modify, and send custom CAN messages.



Figure 5.10: TUCminiCar with the Tiny Can hardware connected.

Lab Setup and Testing Procedure

The setup also involved installing the Gateway Application on a Raspberry Pi and then connecting it to the Tiny CAN device. Here is how the test procedure went:

Tiny CAN Setup and Connection: The Tiny CAN was connected, via USB, with the RaspberryPi. This setup made the Pi act like a CAN bus interface. Tiny CAN software is utilized on a PC for configuring the adapter to send specific CAN messages designed for a dashboard application on the TUCminiCar.

Custom CAN Message Transmission: Tiny CAN allowed the creation and sending of custom CAN messages. Messages were constructed based on the CAN specifications for speed, battery levels, sensor status, and other indicators of the TUCminiCar. After modifying each of these parameters in real time, various states were simulated.

Below is the CAN messages detected by the Gateway Application, which is running on Raspberry Pi, captured, processed, and sent over Wi-Fi to the CE Box via TinyCAN. In real-time, data would flow like this when the Raspberry Pi is connected to the Aurix CAN bus, capturing and forwarding data.

Real-time Updates on the Dashboard: Real-time UI updates were reflected on the various dashboard elements from the data feeds provided by the Gateway Application running on the CE Box. Changes to speed, battery, and proximity sensors correctly reflected on the screen, along with status indicators, so that every response could be identified and validated by the tester from the incoming data.

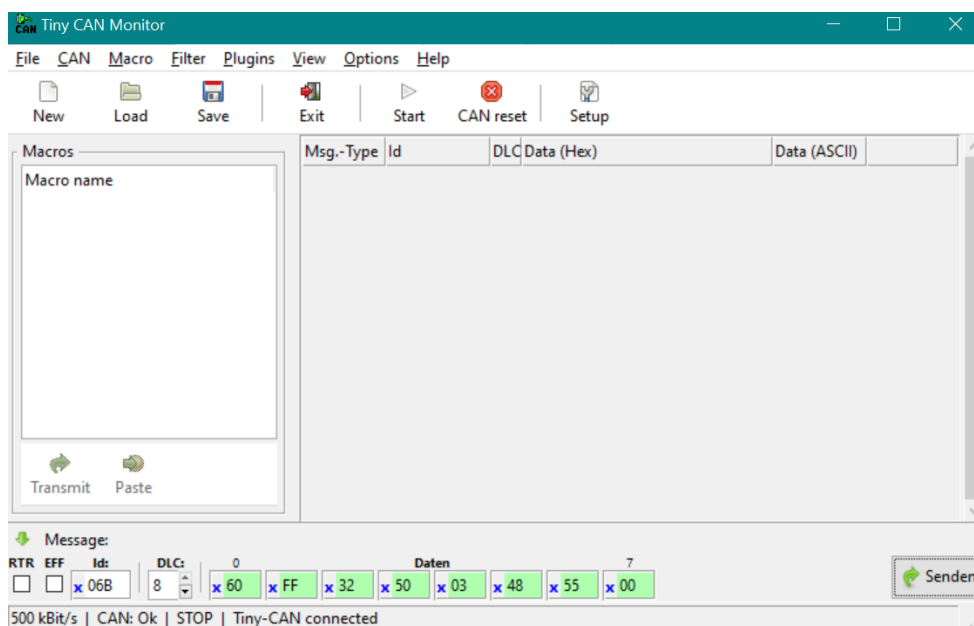


Figure 5.11: Tiny CAN GUI.

This lab validation using Tiny CAN proved important for the fine-tuning of the Gateway Application; in this instance, a controlled environment was enabled where specific conditions could be reproduced to assess each aspect of data flow and dashboard performance. Hence, this proved that such a setup will work on the TUCminiCar.

5.4 Performance Testing

To verify if these CAN data simulation and dashboard applications can function well, performance tests in order to measure latency and responsiveness for this system were performed. This section will discuss in detail how the measurement of latency was performed—from timestamping in various stages of the data transmission-receiving path. The correct measurement of the end-to-end latency between the gateway and dashboard applications will provide insight into whether the solution can meet real-time responsiveness requirements of automotive data systems.

Testing Methodology

The testing methodology was developed to ensure that the data in the developed system would be efficiently transmitted, received, and visualized while estimating and measuring possible latencies. The methodology included the following steps:

1. Data Transmission Setup

Each CAN frame was intercepted by the gateway application running on the Raspberry Pi, which was connected with the Aurix microcontroller via the PiCAN2 interface. These frames were encapsulated into UDP datagrams. Because of the low-latency data transmission requirements for real-time monitoring, UDP was chosen, which is lightweight and connectionless. The gateway application transmits the UDP packets to the dashboard application running on another Raspberry Pi over the Wi-Fi network of the CE Box.

2. Timestamp Logging

To measure and analyze network latency, precise timestamps were logged at critical points in the data flow:

Send Timestamp: The timestamp right after the UDP packet was sent out of the gateway application. This contains a timestamp representing the instance at which the data has just left the gateway device.

Receive Timestamp: This is the timestamp recorded at the instance when the UDP packet was received by the dashboard application. It marked the moment the data arrived and was available for processing.

3. Latency Measurement

The difference in timestamps for send and receive gives a direct measure of network latency.

Calculation of Latency: This gave the exact delays in transmitting data over the network. After getting this latency, one would have been assured that in the real-time update of data on the dashboard application, the performance threshold stays within acceptable limits.

4. Error Analysis and Observations

Additional logging mechanisms captured lost or out-of-order packets during transmission due to network congestion or any other interruptions that may have resulted in the UDP-based communication being affected. Monitoring these occurrences provided insight into how reliable the system was when operating under a variety of test scenarios.

5. Visualization and Validation

After calculating latency, the data received was checked against the expected values to confirm that no data corruption occurred during transmission. Various tests were conducted on the system under different network environments and loads to assess its performance and any potential bottlenecks.

Detailed tracking of data transmission and latency measurements ensured that real-time updates in the dashboard application were reliable. This methodology has been a strong basis for performance validation, while it also underlined optimization possibilities, especially in network setup and packet handling.

Code Implementation

In the gateway application, timestamp immediately after sending each UDP packet to the network was added. This timestamp represents the exact time at which the data was sent.

```
// Send data over UDP and log the send timestamp

udpSocket.writeDatagram(udpPayload, udpHost, udpPort);
qint64 sendTime = QDateTime::currentMSecsSinceEpoch();
qDebug() << "Sent UDP packet at:" << sendTime << " Data:" <<
byteArrayToHexWithSpaces(udpPayload);
```

In the dashboard application, timestamps were recorded as soon as the UDP packet was received. This provides a precise measure of when data arrives at the dashboard for processing.

```
udpSocket->readDatagram(datagram.data(), datagram.size(), &sender,
&senderPort);
```

```
// Log the received timestamp
qint64 receiveTime = QDateTime::currentMSecsSinceEpoch();
qDebug() << "Datagram received at:" << receiveTime << " Data:" <<
byteArrayToHexWithSpaces(datagram);
```

Analysis of Results

By subtracting the send timestamp from the receive timestamp, we calculated the network latency between the gateway and dashboard applications. Here's an example of a latency calculation:

Send Time (Gateway): 1730838094829 ms

Receive Time (Dashboard): 1730838094855 ms

Calculated Latency: receiveTime - sendTime = 26 ms

The latency measurements were consistently low, averaging around 25 ms, which is within the acceptable range for CAN data transmission in many automotive applications [48]. Additionally processing time of the CAN message in the dashboard was also measured by logging the time stamp before and after the processing of the data in the Dashboard application. It was observed that there was 0 or 1 ms difference which conclude that there is barely any delay observed.

The performance testing revealed that the CAN data simulation and dashboard application met the target requirements for latency. The approach of using UDP for data transmission, while not inherently reliable, was appropriate for the low-latency, high-speed requirements of this application. In the future, enhancements could be done by testing other protocols or mechanisms concerning packet losses that would increase reliability, especially in use cases where packet losses could more often happen. In a nutshell, this performance test serves well in proving that the system can support the delivery of real-time CAN data visualization within acceptable latency levels, thus acting as a groundwork basis for further development in automotive HMI systems.

5.5 Observations from testing on the TUCminiCar

The test deployment within the scope of the current thesis included deploying the developed gateway application onto the TUCminiCar demonstrator, which is set to receive and process data from another student-developed software component within the scope of their thesis project for the same TUCminiCar. Tests were carried out in order to check real-time performance of the system, particularly latency between events of detection and its visualization on the dashboard.

It came out during the tests that in all cases, example when an obstacle was detected in real-time, it would always take around 1–2 seconds to show up on the dashboard. In fact, the delay of the signal concerns the processing within the AURIX microcontroller, converting sensor data into a CAN message and sending it onto the CAN bus. This information was on the CAN bus and had been captured by the gateway application and relayed to the dashboard for display.

Further testing, with the data simulation environment, showed that the gateway and dashboard applications used a very minimal amount of processing time. This suggests that the latency experienced begins further earlier in the data pipeline, within the individual software components of the TUCminiCar and deeper into hardware processing by the AURIX controller.

This could form a starting point for further improvements in such a study; thus, this constitutes a limitation of the study. Reducing latency might be achieved by improving processing time within the software modules of the TUCminiCar or reducing delays from the TUCminiCar hardware to the software. In addition, the mechanical nature of the TUCminiCar demonstrator also contributed to the distortion of visualization accuracy, which may imply that further modifications to the hardware could achieve even better real-time performance.

Overall, testing showed that although the gateway and dashboard applications themselves can provide near real-time data processing and visualization, real-time responsiveness in TUCminiCar will still greatly depend on the performance of every software component and hardware module involved in data collection, processing, and CAN transmission. Future work may focus on latency optimization for these components to ensure better real-time feedback.

5.6 Limitation

Limited by hardware, there were several limitations to the TUCminiCar's dashboard application due to the nature of the design. This will address some issues pertaining to processing power, data transfer efficiency, network reliability, and UI scalability.

Hardware constraints: The application for the dashboard was developed using the Raspberry Pi 3B+ due in part to availability. In contrast to more current iterations, the 3B+ has far less processing power and less RAM.

Qt Compatibility: Qt 5.15 is used because it is compatible with the 3B+, and also stable on its operating system. On the other side, new Qt versions-for example, 6.x-

have lots of new features, performances, and a wider range of UI components that would have been useful for the dashboard application.

Using UDP to Send CAN Data Transmission The current design uses to send each CAN signal as an independent UDP packet. Though UDP is lighter-weight and faster for real-time applications, this approach has a number of the following drawbacks:

High Packet Overhead: Since each CAN signal is transmitted as one UDP packet, network load grows linearly with an increasing number of signals. This can lead to a possible packet loss when heavily loaded because UDP doesn't account for acknowledging the packet.

Dependence on the network environment of the CE Box: Dashboard Application depends on the Wi-Fi connection provided by the CE Box for the update of its data from Gateway Application. There is bound to be interference in the network or breaks in connectivity that could cause lagged or dropped real-time data updates. Further testing in various network environments may reveal further optimizations or alternative protocols that can be used to improve its reliability.

Single-screen display: The current design puts everything into one screen, making it user-friendly but also busy when new features or sensors integrate into the platform in the future. This can give way to a more flexible, multi-screen, or modular kind of approach that could make readability and scalability better with more data elements integrated.

These limitations also point out several areas where performances and scalability, both in terms of hardware and efficiency, can be further improved. Upgrading of more robust hardware, enhanced transmission protocols, and consideration of optimizations in data handling will remove these constraints and provide a better dashboard application for further development.

6 Conclusion and Future Work

6.1 Conclusion

The purpose of this thesis was to develop an AUTOSAR-compliant, real-time dashboard application on the QT framework for the TUCminiCar platform. The contribution presented integrates open-source technologies like Raspberry Pi 3B+, CAN bus protocol, and the QT framework to provide an application with a clean and user-friendly interface, which is able to monitor vehicle data such as speed, battery levels, and other sensor statuses. The system is, therefore, specially designed for real-time data acquisition and visualization to support informed decisions by users based on real-time, live vehicle parameters with a view to validating the practicality of modular embedded systems applied in automotive environments. However, this project also unveiled various aspects in which the present setup needs further advancement in both hardware and software segments and improvement in data transmission protocols. These limitations, as identified, could be the basis upon which substantial improvements may be proposed for future versions of this system.

6.2 Future Work

It will be a good basis for the TUC Mini Car real-time data visualization, but there is further room for improvement regarding functionality and usability. Hardware restrictions and single-screen layout, addition of data transmission protocols-deserving improvement in further work at the visualization interface.

Upgrading Hardware and Supported Qt Versions The Raspberry Pi 3B+ constrains processing power and access to new software features. Upgrade to the Raspberry Pi 4 or other systems for better complex visuals and more fluid data processing.

It also provides support for Qt 6.x migration with improved support for the GPU, 3D graphics libraries, and UI components in the system to bring a better user experience and responsiveness to the system.

Visual Effects and 3D Graphics: Future releases of the application, using Qt 6.x, will provide 3D graphics with realistic displays of the vehicle data. The use of Qt 3D and OpenGL can enable interactive visualizations of vehicle models and an environment that would support understanding complex metrics, such as sensor readings and orientation. The improved UI of 3D visualization of vehicle dynamics with clearer data and feedback is meant to increase the user's understanding.

Modular data representation with multi-SCREEN layout: Nowadays, all vehicle data can be accommodated on one screen; however, with the increase in more and more functions, it starts to get cluttered. It will offer the display of data in a multi-screen setup for the key functions of sensor data, diagnostics, and navigation in a modular way, improving readability and leaving space for scalable expansion.

Future development could leverage the Qt modular design framework to extend new data elements and features.

Secure Communication Protocols Furthermore, because the system sends every CAN message as an individual UDP packet, it can give rise to data overhead and congestion in high-load scenarios. Future work should therefore investigate how CAN signals can be aggregated into fewer packets, or using a protocol such as TCP, which may provide improved scalability and reliability.

Data encryption techniques may offer secure data transfers, thus increasing system integrity for safer environments. User-Adaptive Interface: Work can also be done in this dashboard application in the future to let the user select which layouts and data elements will be presented. The touch controls provide for dynamic arrangement in which the layouts change based on user profiles. It will hence give the possibilities to easily adapt the interface to the needs and context of drivers, engineers, and researchers. More testing in various conditions: It should be taken through the different conditions of the automotive and network environment in the next testing to tune its data handling further and to expose performance bottlenecks. Extensive testing will increase the ability of the system to be resilient, efficient, and adaptive in real-world scenarios. With those enhancements, the TUC Mini Car dashboard would be much more modular and ready for future development. Combining these features could make the system a powerful tool for research and development in the area of automotive embedded systems.

7 Bibliography

- [1] H. Mende, "Smart Dashboards: Transforming the Automotive Infotainment System Market," 10 09 2024. [Online]. Available: <https://www.marketresearchintellect.com/blog/smart-dashboards-transforming-the-automotive-infotainment-system-market>. [Accessed 9 11 2024].
- [2] "Classic Platform AUTOSAR," Autosar.org, 2023. [Online]. Available: <https://www.autosar.org/standards/classic-platform>. [Accessed 10 11 2024].
- [3] T. TA, "Exploring the Role of Embedded Systems in Automotive Industry," Travancore Analytics, 19 12 2023. [Online]. Available: Exploring the Role of Embedded Systems in Automotive Industry.
- [4] "Devi, R., Sivakumar, P., & Balaji, R. (2018). AUTOSAR Architecture Based Kernel Development for Automotive Application. International Conference on Intelligent Data Communication Technologies and Internet of Things (ICICI) 2018. <https://doi.org/10.1007>".
- [5] AUTOSAR, "Layered Software Architecture AUTOSAR Classic Platform," AUTOSAR Standard Documentation.
- [6] T. Agarwal, "Embedded Systems Role in Automobiles and Their Working," EIProCus - Electronic Projects for Engineering Students, 8 9 2019. [Online]. Available: <https://www.elprocus.com/embedded-systems-role-in-automobiles/>. [Accessed 10 11 2024].
- [7] D. Kum, G. Park, S.-H. Lee and W. Jung, "AUTOSAR migration from existing automotive software," 2008.
- [8] S. Devi, P. Sivakumar and B. Balaji, "AUTOSAR Architecture Based Kernel Development for Automotive Application," 2018.
- [9] X. Q. Wu, L. L. Li and H. J. Chen, "Realization of CAN Based on Automotive Open System Architecture," 2013.
- [10] A. A. Salunkhe, P. P. Kamble and R. Jadhav, "Design and implementation of CAN bus protocol for monitoring vehicle parameters," 2016.
- [11] T. Rajasekar and K. Bhaskar, "Vehicle Control System using Controller Area Network [Can] Protocol," *International Journal of Communication and Networking System*, vol. 5, no. 1, pp. 68-69, 2015.
- [12] "Future of cyber security in automotive industry," Spyrosoft, 20 02 2024. [Online]. Available: <https://spyro-soft.com/developers/future-of-cyber-security-in-automotive-industry>. [Accessed 10 11 2024].

- [13] S. Woo, H. J. Jo and D. H. Lee, "A Practical Wireless Attack on the Connected Car and Security Protocol for In-Vehicle CAN," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1-14, 2014.
- [14] "About Qt - Qt Wiki," [wiki.qt.io](https://wiki.qt.io/About_Qt), [Online]. Available: https://wiki.qt.io/About_Qt. [Accessed 10 11 2024].
- [15] "Accelerated GUI development with Linux Qt | VOLANSYS," VOLANSYS, 16 11 2016. [Online]. Available: Accelerated GUI development with Linux Qt | VOLANSYS. [Accessed 10 11 2024].
- [16] "Qt Creator Documentation," [Doc.qt.io](https://doc.qt.io), 2024. [Online]. Available: <https://doc.qt.io/qtcreator/index.html>. [Accessed 10 11 2024].
- [17] "Qt for Windows - Requirements | Qt 5.15," [Doc.qt.io](https://doc.qt.io), 2015. [Online]. Available: <https://doc.qt.io/qt-5/windows-requirements.html>. [Accessed 10 11 2024].
- [18] A. Schmidt and B. Pfleging, "Automotive User Interfaces," *it - Information Technology*, vol. 54, no. 4, pp. 155-156, 2012.
- [19] S. Schneegaß, B. P. D. Kern and A. Schmidt, "Support for modeling interaction with automotive user interfaces," 2011.
- [20] u. Auto, "The near future of in-car HMI | ustwo Insights," 2014. [Online]. Available: <https://ustwo.com/blog/the-near-future-of-in-car-hmi/>. [Accessed 10 11 2024].
- [21] N. P. M, "Acsia | Automotive User Experience with Digital Cockpits," Acsia Technologies PVT LTD, 10 6 2024. [Online]. Available: <https://www.acsiatech.com/insights/the-future-of-automotive-user-experience-building-the-ultimate-digital-cockpit/>. [Accessed 10 11 2024].
- [22] "The Future of User Interfaces: Voice and Gesture Control in Software Development - wonderIT," wonderIT, 30 8 2023. [Online]. Available: <https://wonderit.io/the-future-of-user-interfaces/>. [Accessed 10 11 2024].
- [23] G. Prabhakar and P. Biswas, "A Brief Survey on Interactive Automotive UI," *Transportation Engineering*, vol. 6, 2021.
- [24] "Qt Automotive Software Development Tooling," [Www.qt.io](https://www.qt.io), 2024.
- [25] "Elektrobit enhances flexibility, capability, and ease of use of HMI development software EB GUIDE, adds support for Raspberry Pi – Elektrobit," Elektrobit, 10 12 2021. [Online]. Available: <https://www.elektrobit.com/newsroom/enhanced-flexibility-capability-eou-of-hmi-development-software-eb-guide/>. [Accessed 10 11 2024].
- [26] "Unity Technologies Increases Commitment to Automotive with Dedicated Team, Industry Bundle, and Two-Day Unity AutoTech Summit," Unity, 2018. [Online]. Available: <https://unity.com/news/unity-technologies-increases-commitment->

- automotive-dedicated-team-industry. [Accessed 10 11 2024].
- [27] J. Larkin, "Open source platform for use in, in-vehicle infotainment or IVI products," 2009. [Online]. Available: <https://www.ai-online.com/2009/10/open-source-platform-for-use-in-in-vehicle-infotainment-or-ivi-products/>. [Accessed 10 11 2024].
- [28] teampivotelec, "25 Top UI Design Software Tools for User Interface Engineers," Pannam, 13 11 2014. [Online]. [Accessed 10 11 2024].
- [29] "DENSO and BlackBerry Launch Integrated Automobile HMI Platform," Blackberry.com, 2019. [Online]. Available: <https://www.blackberry.com/us/en/company/newsroom/press-releases/2019/denso-and-blackberry-launch-integrated-automobile-hmi-platform>. [Accessed 10 11 2024].
- [30] "Automotive & EV Vehicle GUI Development Software | Crank AMETEK," Cranksoftware.com, 2024. [Online]. Available: <https://www.cranksoftware.com/industries/automotive>. [Accessed 10 11 2024].
- [31] "CGI Studio 3.10: faster, smarter, more direct," CGI Studio, 23 6 2021. [Online]. Available: <https://cgistudio.at/cgi-studio-3-10-2/>. [Accessed 10 11 2024].
- [32] "Kanzi Studio: Is it Really Transforming Automotive UI Design? | GlobalLogic," GlobalLogic, 23 12 2024. [Online]. Available: <https://www.globallogic.com/insights/blogs/is-kanzi-really-transforming-ui-design/>. [Accessed 10 11 2024].
- [33] M. Jiang, "Design of Electric Vehicle Control System Based on CAN Bus," *2022 IEEE 5th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, 2022.
- [34] R. M. Patil, K. P. Chethan, R. R. N. H K and S. R. , "Infotainment System Using CAN Protocol and System on Module with Qt Application for Formula-Style Electric Vehicles," *Advances in intelligent systems and computing*, pp. 215-226, 2020.
- [35] P. D. D. h. W. Hardt, M. S. O. Khan and H. Aljaere, "Adaptive User Interface for Automotive," Chemnitz, 2021.
- [36] P. D. D. W. Hardt, D. W. Wilke and B. H. Santoso, "Development of a Framework for 3D," Technische Universität Chemnitz, Chemnitz, 2015.
- [37] H. SANELL and G. SAMUELSSON, "In Vehicle Infotainment Demonstrator," Göteborg, Sweden, 2011.
- [38] Y.-j. DI, N. JIN and L.-g. WANG, "Design of Virtual Instrument Interface Based on Embedded Linux+QT," *DEStech Transactions on Computer Science and*

Engineering, 2022.

- [39] L.-Y. K.-Y. H.-. P. and L.-J. , "The Research of Qt/Embedded and Embedded Linux application in the Intelligent Monitoring System control," 2010.
- [40] H. Lönn, "Project within Vehicle Development: Report on AUTOSAR and QT-Based Visualization,," *Vinnova*, 2017.
- [41] I. T. AG, "Motor Control AURIX™ TC387 - Infineon Technologies," Infineon.com, 2024. [Online]. Available: <https://www.infineon.com/cms/en/product/promopages/AURIX-microcontroller-boards/application-boards/motor-control/>. [Accessed 10 11 2024].
- [42] "Raspberry Pi CAN Bus Integration: An In-Depth Working Guide," AutoPi.io, [Online]. Available: <https://www.autopi.io/blog/raspberry-pi-can-bus-explained/>. [Accessed 10 11 2024].
- [43] "OBD-II Data Logging With Raspberry Pi And PiCAN2 CAN Bus Interface," Copperhill, 2019. [Online]. Available: <https://copperhilltech.com/blog/obdii-data-logging-with-raspberry-pi-and-pican2-can-bus-interface/>. [Accessed 10 11 2024].
- [44] Maxbotix, "Maxbotix," 27 11 2023. [Online]. Available: <https://maxbotix.com/products/mb1603>. [Accessed 27 11 2024].
- [45] D. L. L. -. W. Wiki, Waveshare.com, 2024. [Online]. Available: https://www.waveshare.com/wiki/DTOF_LIDAR_LD19. [Accessed 27 11 2024].
- [46] J. Nine, S. Saleh, O. Khan and W. Hardt, "Traffic Light Sign Recognition for Situation Awareness using Monocular Camera," *Symposium ISCSET 2019*, 2019.
- [47] "Übersicht Tiny-Can | MHS Online-Shop," Mhs-elektronik.de, [Online]. Available: https://mhs-elektronik.de/index.php?module=content&action=show&page=tinycan_uebersicht. [Accessed 11 11 2024].
- [48] M. Vasilevski and I. Píša, "CAN Bus Latency Test Automation for Continuous Testing and Evaluation," 2022.
- [49] M. Stolpe, *AUTOSAR Application Visualization Degree Project, BSc in Computer Engineering*, 2014.

Appendix A – Qt framework

Qt 5.15 Installation on Raspberry Pi 3B+

This appendix is a step-by-step installation of Qt 5.15 on Raspberry Pi 3B+ running the Raspberry Pi OS, supported with detailed commands. To install the Qt framework, the subsequent commands had to be issued to ensure that all packages with their dependencies were installed. Step-by-Step Installation

1. System Update and Upgrade.

Begin by updating and upgrading the package list to ensure the latest versions are used.

```
sudo apt update
sudo apt upgrade
```

2. Install Qt Base and Essential Libraries

The following command installs the core Qt libraries and related dependencies required for Qt applications:

```
sudo apt-get install -y qtbase5-dev qt5-qmake libqt5webengine-data
libboost-all-dev libudev-dev libinput-dev libinput-dev libts-dev
libmtdev-dev libfontconfig1-dev libssl-dev libdbus-1-dev libglib2.0-
dev libxkbcommon-dev libegl1-mesa-dev libgbm-dev libgles2-mesa-dev
mesa-common-dev libasound2-dev libpulse-dev gstreamer1.0-omx
libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev gstreamer1.0-
alsa libsrtcp2-1 "^libxcb.*" flex bison libxslt-dev ruby gperf libbz2-
dev libcups2-dev libatkmm-1.6-dev libxi6 libxcompositel libfreetype6-
dev libicu-dev libsqlite3-dev libxslt1-dev libavcodec-dev
libavformat-dev libswscale-dev freetds-dev libsqlite0-dev libpq-dev
libiodbc2-dev firebird-dev libgst-dev libxext-dev libxcb1 libxcb1-dev
libx11-xcb1 libx11-xcb-dev libxcb-keysyms1 libxcb-keysyms1-dev
libxcb-image0 libxcb-image0-dev libxcb-shm0 libxcb-shm0-dev libxcb-
icccm4 libxcb-icccm4-dev libxcb-glx0-dev libxi-dev libdrm-dev libssl-
dev libxcb-xinerama0 libxcb-xinerama0-dev libatspi2.0-0 libxcb-sync1
libxcb-sync-dev libxcb-render-util0 libxcb-render-util0-dev libxcb-
xfixes0-dev libxrender-dev libxcb-shape0-dev libxcb-randr0-dev
libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev libgstreamer-
plugins-bad1.0-dev gstreamer1.0-plugins-base gstreamer1.0-plugins-
good gstreamer1.0-plugins-bad gstreamer1.0-plugins-ugly gstreamer1.0-
libav gstreamer1.0-tools gstreamer1.0-x gstreamer1.0-alsa
gstreamer1.0-gl gstreamer1.0-gtk3 gstreamer1.0-qt5 gstreamer1.0-
pulseaudio
```

3. Install Qt Development Tools

These additional tools include Qt Creator (IDE) and various Qt modules, useful for developing and testing applications:

```
sudo apt-get install -y qt5-qmake qttools5-dev-tools qtcreator
qtdeclarative5-dev qtquickcontrols2-5-dev libqt5serialport5
libqt5serialport5-dev libqt5charts5 libqt5charts5-dev qml-module-
```

```
qtquick-controls2    qml-module-qtquick-dialogs    qml-module-qtquick-  
virtualkeyboard    qml-module-qtquick-*    qml-module-qt-*
```

4. Verify Installation

Confirm the Qt version installed by running:

```
qmake --version
```

This completes the setup of Qt 5.15 on the Raspberry Pi 3B+, providing a foundation for development and deployment of Qt applications.

5. Configure Qt Creator for Qt 5.15

After installing Qt 5.15 and Qt Creator, follow these steps to configure the development kit within Qt Creator:

Open Qt Creator

Go to Tools > Options (or Qt Creator > Preferences on macOS)

Select "Qt Versions" under the "Kits" section

Click "Add" to add a new Qt version

Browse to the qmake executable for Qt 5.15, typically located at:

```
<Qt install path>/5.15.x/gcc_64/bin/qmake
```

Give it a name like "Qt 5.15.x"

Configure Compiler

In the same Options/Preferences window, select "Compilers"

Ensure their compiler (e.g. GCC, MinGW, MSVC) is detected

If not, click "Add" to manually add it

Set Up Kit

Select "Kits" in the sidebar

Click "Add" to create a new kit

Give it a name like "Desktop Qt 5.15.x"

Set the following options:

Device type: Desktop

Device: Local PC

Compiler: Select your compiler

Qt version: Select the Qt 5.15 version you added.

Qt mkspec: Leave default

Click "Apply" then "OK" to save the kit configuration.

Verify Configuration

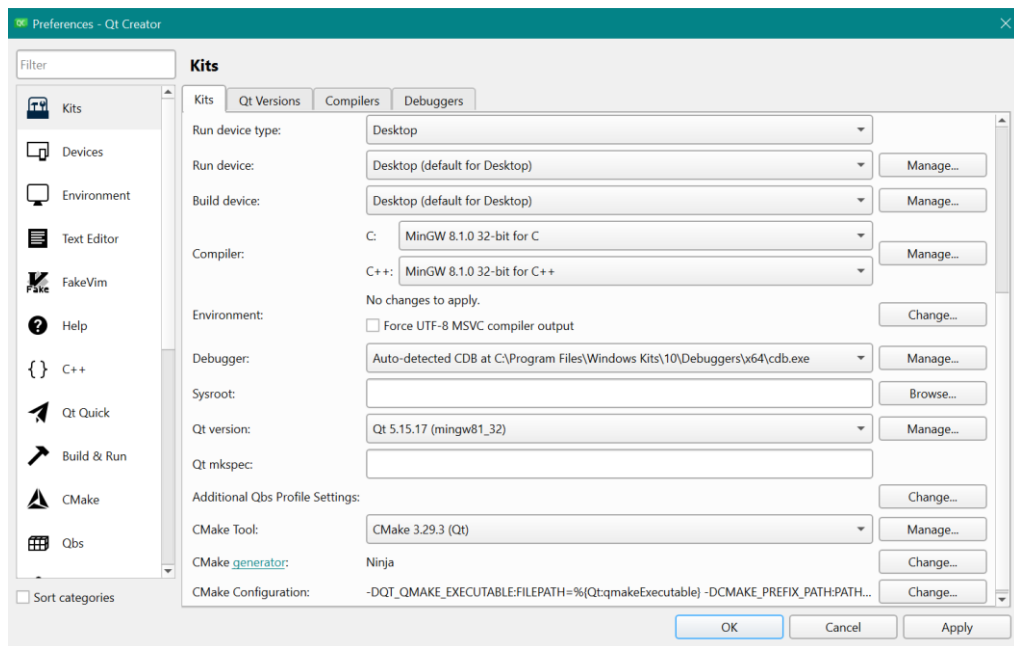
Create a new Qt Widgets Application project.

In the Kit Selection step, choose your new Qt 5.15 kit.

Finish creating the project.

Build and run to verify everything is working correctly.

With these steps completed, you should now have Qt Creator properly configured to develop with Qt 5.15. The new kit will be available when creating or opening Qt projects.



QT it Configuration.

Step 6: Verify Kit Configuration

Open a test project or create a new project to verify that the Qt 5.15 kit is correctly configured. Compile and run a sample application to ensure everything is functioning as expected.

Appendix B- Application Configuration

Appendix B – Configuring the Application

This appendix describes how to deploy and test the application on various configuration settings: running the simulator and the dashboard all on PC, running the dashboard on a Raspberry Pi with the simulator on a PC, and running the dashboard on an RPI connected to a MiniCar.

1) Setting up a PC - Simulator and Dashboard on the PC

This is the configuration for which both DataSimulator and Dashboard are running at the same PC. The application uses the virtual CAN interface, vcan0 to simulate the CAN messages.

Transfer code to a Linux machine and open it via QT creator.

Prepare the Virtual CAN Interface:

Load the vcan module:

```
sudo modprobe vcan
```

Create the virtual CAN interface (vcan0):

```
sudo ip link add dev vcan0 type vcan
```

Bring up the virtual CAN interface:

```
sudo ip link set up vcan0
```

Configure DataGateway and Dashboard:

In DataGateway:

```
QHostAddress udpHost = QHostAddress::LocalHost;
```

In Dashboard:

```
udpSocket->bind(QHostAddress::LocalHost, 12345);
```

Build and Run:

Open the DataSimulator, DataGateway and Dashboard project and run it.

2) PC with Raspberry Pi Setup (Simulator on PC, Dashboard on RPI)

In this setup, the Dashboard runs on an RPI, while the DataSimulator and DataGateway run on a PC. This setup allows testing the dashboard on the actual RPI hardware with simulated data sent over the network.

Raspberry Pi (RPI) Setup:

Transfer the Dashboard code to the RPI.

In Dashboard:

```
udpSocket->bind(QHostAddress::Any, 12345); // Allows listening to any IP
```

Build and run the Dashboard project on the RPI.

PC Setup:

Find the IP address of the RPI (use `ping` to verify connectivity).

In DataGateway:

```
QHostAddress udpHost = QHostAddress("IP_ADDR_OF_RPI");
```

Prepare the Virtual CAN Interface (steps described previously)

Open the **DataSimulator** and **DataGateway** project, build, and run it.

3) PC with MiniCar Setup (Dashboard on RPI, Physical CAN Connection)

In this setup, the Dashboard application runs on the RPI, which connects to the MiniCar via a physical CAN interface (e.g., `can0`). The PC is used to run the DataGateway and DataSimulator applications, with the CAN interface configured on both PC and RPI.

Raspberry Pi (RPI) Setup:

Transfer the Dashboard code to the RPI.

In Dashboard:

```
udpSocket->bind(QHostAddress::Any, 12345); // Allows listening to any IP
```

Build and run the Dashboard project on the RPI.

PC Setup for CAN Interface:

On the PC, configure the CAN interface with the following commands (assuming the CAN interface is connected via USB):

```
sudo modprobe can
sudo modprobe can_raw
sudo modprobe can_dev
sudo ip link set can0 up type can bitrate 500000
```

Configure DataGateway for RPI IP and Physical CAN Interface:

In DataGateway:

Replace the below with the actual IP address of the RPI:

```
QHostAddress udpHost = QHostAddress("IP_ADDR_OF_RPI");
```

Replace vcan0 with can0:

```
QCanBusDevice* canDevice = QCanBus::instance()->createDevice("socketcan",  
"can0");
```

Build and Run the DataGateway application.

By following these configurations, users can deploy and test the application in various environments, allowing flexibility for both simulation and real-world testing scenarios. This appendix serves as a guide to ensure proper setup for each testing configuration.

Appendix C – Code Snippets

CAN-to-UDP Transmission Logic

This code reads CAN messages from the can0 interface and transmits them over UDP to the dashboard application. It prepares a UDP payload containing the CAN frame ID and data payload, which is then sent to the specified IP and port.

```
QObject::connect(canDevice, &QCanBusDevice::framesReceived, [&]() {
    while (canDevice->framesAvailable()) {
        QCanBusFrame frame = canDevice->readFrame();

        // Prepare UDP payload with CAN frame ID and payload
        QByteArray udpPayload;
        udpPayload.append(static_cast<char>(frame.frameId()));
        udpPayload.append(frame.payload());
        udpSocket.writeDatagram(udpPayload, udpHost, udpPort);
        qDebug() << byteArrayToHexWithSpaces(udpPayload);
    }
});
```

Signal Handling for Safe Application Exit

To safely exit the Gateway application, this code captures the CTRL+C interrupt signal. The signal triggers an exit message and halts the application, ensuring a clean shutdown process.

```
void signalHandler(int signal) {
    exitRequested = true;
    std::cout << "CTRL+C detected, exiting..." << std::endl;
}

signal(SIGINT, signalHandler);
```

CAN Device Setup and Initialization

This snippet establishes a connection to the can0 interface on the Raspberry Pi. It checks if the CAN device was successfully created and connected, outputting an error message if the setup fails.

```
QCanBusDevice* canDevice = QCanBus::instance()->createDevice("socketcan",
"can0");
if (!canDevice || !canDevice->connectDevice()) {
    qDebug() << "Failed to connect to CAN device.";
    return -1;
}
```

Project Configuration File (DataGateway.pro)

The DataGateway.pro file specifies the Gateway application's dependencies, including network and serialbus for CAN communication, as well as console mode for terminal-based operation.

```
QT += core network serialbus
CONFIG += c++11 console

SOURCES += main.cpp
```

Battery Gauge Component (HorizontalGauge.qml)

This QML code creates a horizontal gauge to visualize battery voltage, with warning thresholds and color changes for critical battery levels.

```
Rectangle {
    id: hgTextObj
    property string myText:"Battery Voltage: "
    property string myTextColor:"black"
    x: 15
    y: 47
    Text {
        text: hgTextObj.myText + _displayValue + "%"
        color: hgTextObj.myTextColor
    }
}
```

Main Dashboard Logic for Indicator Control (main.qml)

This QML function updates the dashboard's visual elements based on CAN signals, such as switching between brake light indicators.

```
function updateBrakeLightIndicator(status, brakeLight) {
    vehicle._imgPath = brakeLight ? "/Images/Car_with_brake_lights_on.png"
: "/Images/Car_with_brake_lights_off.png";
}
```

UDP Receiver Functionality (UdpReceiver.cpp)

The UDP receiver handles incoming UDP datagrams, interpreting the CAN data and emitting signals to update the dashboard components.

```
void UdpReceiver::processPendingDatagrams() {
    while (udpSocket->hasPendingDatagrams()) {
        QByteArray datagram;
        udpSocket->readDatagram(datagram.data(), datagram.size());
        emit dataReceived(datagram);
    }
}
```

Appendix C – QR Codes



Thesis Implementation



Dashboard in TUCminiCar



This report - except logo Chemnitz University of Technology - is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made. The images or other third party material in this report are included in the report's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the report's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/>.

Chemnitzer Informatik-Berichte

In der Reihe der Chemnitzer Informatik-Berichte sind folgende Berichte erschienen:

- CSR-21-01** Marco Stephan, Batbayar Battseren, Wolfram Hardt, UAV Flight using a Monocular Camera, März 2021, Chemnitz
- CSR-21-02** Hasan Aljzaere, Owes Khan, Wolfram Hardt, Adaptive User Interface for Automotive Demonstrator, Juli 2021, Chemnitz
- CSR-21-03** Chibundu Ogbonnia, René Bergelt, Wolfram Hardt, Embedded System Optimization of Radar Post-processing in an ARM CPU Core, Dezember 2021, Chemnitz
- CSR-21-04** Julius Lochbaum, René Bergelt, Wolfram Hardt, Entwicklung und Bewertung von Algorithmen zur Umfeldmodellierung mithilfe von Radarsensoren im Automotive Umfeld, Dezember 2021, Chemnitz
- CSR-22-01** Henrik Zant, Reda Harradi, Wolfram Hardt, Expert System-based Embedded Software Module and Ruleset for Adaptive Flight Missions, September 2022, Chemnitz
- CSR-23-01** Stephan Lede, René Schmidt, Wolfram Hardt, Analyse des Ressourcenverbrauchs von Deep Learning Methoden zur Einschlagslokalisierung auf eingebetteten Systemen, Januar 2023, Chemnitz
- CSR-23-02** André Böhle, René Schmidt, Wolfram Hardt, Schnittstelle zur Datenakquise von Daten des Lernmanagementsystems unter Berücksichtigung bestehender Datenschutzrichtlinien, Januar 2023, Chemnitz
- CSR-23-03** Falk Zaumseil, Sabrina Bräuer, Thomas L. Milani, Guido Brunnett, Gender Dissimilarities in Body Gait Kinematics at Different Speeds, März 2023, Chemnitz
- CSR-23-04** Tom Uhlmann, Sabrina Bräuer, Falk Zaumseil, Guido Brunnett, A Novel Inexpensive Camera-based Photoelectric Barrier System for Accurate Flying Sprint Time Measurement, März 2023, Chemnitz
- CSR-23-05** Samer Salamah, Guido Brunnett, Sabrina Bräuer, Tom Uhlmann, Oliver Rehren, Katharina Jahn, Thomas L. Milani, Günter Daniel Rey, NaturalWalk: An Anatomy-based Synthesizer for Human Walking Motions, März 2023, Chemnitz
- CSR-24-01** Seyhmus Akaslan, Ariane Heller, Wolfram Hardt, Hardware-Supported Test Environment Analysis for CAN Message Communication, Juni 2024, Chemnitz

Chemnitzer Informatik-Berichte

- CSR-24-02** S. M. Rizwanur Rahman, Wolfram Hardt, Image Classification for Drone Propeller Inspection using Deep Learning, August 2024, Chemnitz
- CSR-24-03** Sebastian Pettke, Wolfram Hardt, Ariane Heller, Comparison of maximum weight clique algorithms, August 2024, Chemnitz
- CSR-24-04** Md Shoriful Islam, Ummay Ubaida Shegupta, Wolfram Hardt, Design and Development of a Predictive Learning Analytics System, August 2024, Chemnitz
- CSR-24-05** Sopoluchukwu Divine Obi, Ummay Ubaida Shegupta, Wolfram Hardt, Development of a Frontend for Agents in a Virtual Tutoring System, August 2024, Chemnitz
- CSR-24-06** Saddaf Afrin Khan, Ummay Ubaida Shegupta, Wolfram Hardt, Design and Development of a Diagnostic Learning Analytics System, August 2024, Chemnitz
- CSR-24-07** Túlio Gomes Pereira, Wolfram Hardt, Ariane Heller, Development of a Material Classification Model for Multispectral LiDAR Data, August 2024, Chemnitz
- CSR-24-08** Sumanth Anugandula, Ummay Ubaida Shegupta, Wolfram Hardt, Design and Development of a Virtual Agent for Interactive Learning Scenarios, September 2024, Chemnitz
- CSR-25-01** Md. Ali Awlad, Hasan Saadi Jaber Aljzaere, Wolfram Hardt, AUTO-SAR Software Component for Atomic Straight Driving Patterns, März 2025, Chemnitz
- CSR-25-02** Billava Vasantha Monisha, Hasan Saadi Jaber Aljzaere, Wolfram Hardt, Automotive Software Component for QT Based Car Status Visualization, März 2025, Chemnitz

Chemnitzer Informatik-Berichte

ISSN 0947-5125

Herausgeber: Fakultät für Informatik, TU Chemnitz
Straße der Nationen 62, D-09111 Chemnitz