Communication Framework for Electrified Off-Road Vehicles: A Case Study on the HHEA Compact Track Loader

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Sujeendra Ramesh

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Perry Li

May, 2025

Acknowledgements

This work is supported by the U.S. Department of Energy Office of Energy Efficiency and Renewable Energy's (EERE) Vehicle Technologies Office (VTO) under grant number DE-EE0009875. Their support has been instrumental in enabling the research and development presented in this thesis.

I would like to express my heartfelt gratitude to all the individuals who have contributed, both directly and indirectly, to the completion of my thesis.

First and foremost, I would like to extend my sincere thanks to my advisor, Professor Perry Li, for his unwavering guidance, encouragement, and insightful feedback throughout this research. His expertise has been instrumental in shaping the direction of my work.

Dedication

This thesis is dedicated to my family, whose love, encouragement, and sacrifices have been my greatest source of strength. To my parents, for their unwavering belief in me and for providing me with the opportunity to pursue my dreams. I also dedicate this work to my friends, whose support and motivation have been invaluable throughout this journey.

To all the mentors and peers who have guided me along the way, I am deeply grateful for your wisdom and inspiration. This achievement would not have been possible without all of you.

Abstract

This thesis presents the development of a generalized, open-source software framework designed to support Controller Area Network (CAN) communication and visualization in electrified off-road vehicles. With the rising momentum toward electrification in heavy-duty and off-highway equipment, off-road platforms face unique challenges—including rugged operating conditions, high power demands, distributed control systems, and the integration of multiple electric actuators. These vehicles typically demand precise coordination and robust communication between subsystems such as traction, hydraulics, and auxiliary functions.

To address these challenges, this work proposes a modular and extensible framework that is protocol-agnostic and configuration-driven. It supports decoding and encoding CAN messages via DBC files, enabling seamless integration across varied architectures. At its core, the framework incorporates a JSON-defined, condition-based finite state machine (FSM) responsible for managing vehicle states, enabling event- and threshold-based transitions, and handling fault detection in a thread-safe manner. Its lightweight and reusable design make it ideal for embedded software on vehicle control units (VCUs).

The framework is validated through a case study on an electrified Compact Track Loader utilizing a Hybrid Hydraulic-Electric Architecture (HHEA), a research vehicle featuring five independently controlled electric motors in concert with a hydraulic common pressure rail system. Here, the hydraulic common pressure rails provide the majority of the power whereas the four of the electric motors modulate that power. The electric motors consist of two small traction motors, mounted in tandem with the left and right hydraulic propel motors, that deliver the modulating torques; two additional motors connected to two hydraulic pump/motors, which regulate pressure—and thereby force—for the lift and tilt hydraulic functions; and a primary electric motor that drives the Digital Displacement Pump (DDP), responsible for supplying flow to the common pressure rail. All hydraulic actuators operate in coordination with this common pressure rail system, which is dynamically supplied by the DDP based on real-time system demand. Coordinating these high-voltage subsystems requires a real-time, fault-resilient control strategy with reliable communication across a J1939-based CAN

network. The vehicle's architecture also includes a power distribution unit (PDU), battery management system (BMS), and several supporting ECUs, all integrated via the proposed framework.

To aid in development and testing, a Qt-based dashboard is developed as part of the framework. This tool enables real-time monitoring and visualization of CAN signals, execution of control commands (such as vehicle start/stop and motor enable), and live diagnostic feedback during both bench and Hardware-in-the-Loop (HIL) testing. The dashboard supports additional features such as fault injection, data logging, and dynamic signal plotting, allowing developers to iterate quickly and verify functionality across diverse testing conditions.

The framework's open-source nature, combined with thorough documentation and modular software design, encourages adoption and customization across other electrified off-highway platforms. By abstracting protocol-specific logic and unifying control, diagnostics, and visualization under a single architecture, this work provides a scalable and reusable foundation for CAN-based communication in off-road electric vehicles.

Through the successful implementation on the HHEA Compact Track Loader, this thesis demonstrates the feasibility and robustness of the proposed approach in managing multi-motor coordination, diagnostic visibility, and flexible software architecture in the context of next-generation electrified off-road systems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Chapter Overview

The off-road vehicle industry is experiencing a paradigm shift with the increasing adoption of electrification technologies. This transition is driven by stricter environmental regulations, the need for improved energy efficiency, and growing interest in reducing maintenance and operational costs. Electrified off-road platforms—including compact track loaders, excavators, and agricultural machinery—pose distinct challenges due to their demanding operational environments, low-volume customization, and the need to support a wide range of auxiliary functions.

Unlike on-road vehicles, which typically use a single electric motor and benefit from standardized communication protocols, off-road vehicles require multiple degrees of freedom to perform tasks like lifting, tilting, and propulsion. Electrifying these systems demands several high-power electric motors—not just for driving, but also for operating hydraulic pumps and auxiliary implements. Each motor functions as an independent subsystem, requiring precise control and robust communication.

The Hybrid Hydraulic-Electric Architecture (HHEA) used in this work exemplifies this complexity, integrating five electric motors to coordinate propulsion and hydraulic actuation. This highlights the need for a scalable and fault-resilient communication framework to manage the distributed control architecture of electrified off-road vehicles.

In this context, Controller Area Network (CAN) communication protocols—particularly SAE J1939 (see Appendix A.1.1)—serve as the backbone for real-time data exchange between Electronic Control Units (ECUs), such as inverters, battery management systems (BMS), and the vehicle control unit (VCU). However, developers working on electrified off-road systems often face substantial barriers, including the need to manage protocol-specific details, decode signals from proprietary DBC files (see Appendix A.4), design human-machine interfaces (HMI), and validate system behavior in both Hardware-in-the-Loop (HIL) and real-world environments.

To address these challenges, this research introduces a modular, DBC-driven framework for CAN communication and visualization that abstracts low-level protocol handling and facilitates rapid development and testing. The framework supports condition-based logic, real-time visualization, and extensibility across multiple vehicle platforms.

To develop this framework, an electrified compact track loader that uses the HHEA [1] is used as a case study.

This chapter introduces the motivation behind the research, outlines the specific research objectives, and provides an overview of the structure and contributions of the thesis.

1.2 Research Objectives

The primary objective of this research is to design and implement a modular, protocol-agnostic framework for CAN communication and signal visualization. This framework is intended to simplify development, testing, and validation across automotive and off-road applications by abstracting low-level protocol details and enabling DBC-based configuration. As part of this effort, the framework will be implemented and validated on an electrified compact track loader (CTL) to demonstrate its effectiveness in a real-world application.

The specific objectives of this thesis are as follows:

- Develop a generic, extensible framework capable of supporting multiple CAN protocols, including but not limited to SAE J1939, through DBC file parsing.
- Implement a condition-based, JSON-driven finite state machine (FSM) for managing system states, transitions, and fault handling logic.

- Create an intuitive Qt-based human-machine interface (HMI) for real-time CAN signal monitoring, control, and diagnostics.
- Validate the framework using Hardware-in-the-Loop (HIL) testing methodology.
- Demonstrate the application of the framework through a real-world case study on an electrified compact track loader using J1939 protocol.
- Release the framework as open-source along with comprehensive documentation to support adoption and customization.

1.3 Thesis Outline

The remainder of this thesis is organized as follows:

- **Chapter 2: Background and Literature Review**
  This chapter introduces the electrified compact track loader prototype being developed at the University of Minnesota and explains the need for robust communication across its subsystems using the SAE J1939 protocol. It also reviews existing tools, communication frameworks, and visualization systems, identifying the research gap in creating a modular, testable, and open-source communication framework for CAN-based vehicle platforms.
- **Chapter 3: System Architecture and Framework Design**
  This chapter describes the architecture of the proposed CAN communication framework. It details the design philosophy, protocol-agnostic structure, DBC-based configuration, and the JSON-driven finite state machine (FSM) developed to handle signal-based transitions, fault detection, and safety logic.
- **Chapter 4: Dashboard Design and Implementation**
  This chapter presents the implementation of the Qt-based dashboard that acts as the human-machine interface (HMI) for the framework. It explains the design choices, signal decoding, user interaction, and how real-time CAN data is visualized and controlled through the dashboard.

- **Chapter 5: Hardware-in-the-Loop (HIL) Testing**
  This chapter discusses the HIL setup used to validate the framework's real-time behavior in a simulated environment. It includes details on the testbench, signal emulation, fault injection strategies, and how the FSM and CAN message timing were verified under controlled conditions.

- **Chapter 6: Vehicle Integration and Real-World Testing**
  This chapter describes the deployment of the framework on the electrified compact track loader. It outlines the vehicle integration process, signal mapping, live communication performance, and how the dashboard and FSM responded in actual operating scenarios.

- **Chapter 7: Results and Analysis**
  This chapter evaluates the performance of the framework under both HIL and real-world testing. It includes analysis of CAN message cycle timing, CPU and memory usage, FSM execution latency, and dashboard frame rate and responsiveness.

- **Chapter 8: Conclusion and Future Work**
  This chapter summarizes the key contributions of the thesis, reflects on the framework's strengths and limitations, and discusses future improvements, including multi-protocol support, deeper diagnostics integration, and expanded visualization capabilities.

Chapter 2

Background and Literature Review

2.1 Chapter Overview

This chapter introduces the electric compact track loader prototype using a Hybrid Hydraulic-Electric Architecture (HHEA) [1] that is being developed at the University of Minnesota. It elaborates on the challenges of communication and coordination between subsystems in such high-power off-road machines, focusing on the role of CAN-based communication protocols, particularly SAE J1939. The chapter also presents a review of the existing literature on vehicle communication frameworks, visualization tools, and identifies the research gap this thesis addresses: the development of a modular and testable communication and visualization framework.

2.2 Prototype Vehicle: System Overview and Architecture

The prototype vehicle developed for this thesis is an electrified compact track loader that embodies the Hybrid Hydraulic-Electric Architecture (HHEA) concept. This architecture strategically combines the high power density and durability of hydraulics with the controllability and efficiency of electric actuation. Central to the HHEA design is a

set of multiple discrete pressure rails, each maintained at a different hydraulic pressure level. These pressure rails function as quantized sources of hydraulic energy, enabling energy-efficient operation across diverse actuation tasks.

Each hydraulic actuator or motor in the system is connected to a valve manifold capable of linking its two ports to any of the four common pressure rails. These connections are dynamically selected by solenoid-operated valves under the control of the Vehicle Control Unit (VCU). This configuration results in a discrete set of 4^2 pressure pairings, allowing for a wide range of achievable actuator forces or hydraulic motor torques. For instance, when the loader arm is commanded to raise, the VCU selects a high-to-low pressure rail combination and opens the corresponding valves to route hydraulic fluid accordingly. The actuator's motion is thus governed by the differential pressure between the selected rails.

To maintain and replenish these pressure rails, a Digital Displacement Pump (DDP) is employed. The DDP is driven by a Danfoss Editron EM-PMI300-T310 electric motor controlled by an Editron EC-C1200-450 inverter. Rather than modulating flow directly at the actuator level, the DDP dynamically fills and stabilizes each pressure rail based on overall system demands. This centralized pressure generation approach ensures that energy is distributed where and when needed, while avoiding unnecessary throttling losses.

In addition to the DDP, the system incorporates four smaller electric motor-generator units (M/Gs), each consisting of a Parker GVM210 motor paired with a Sevcon HVLP (High Voltage Low Power) inverter. These motors are mounted on hydraulic pump/-motor assemblies and are dedicated to key vehicle functions—specifically, left and right propulsion as well as the lift and tilt work functions. Their role is to finely modulate the hydraulic flow or torque seen at the actuator or motor, providing continuous control within the discrete force or torque steps defined by the pressure rail combinations.

For lift and tilt, the small electric motors operate in conjunction with their respective hydraulic units to either boost or absorb pressure, effectively modifying the force applied by the actuator. For propulsion, the motors are coupled in tandem with the hydraulic drive motors and can add to or subtract from the torque produced by the hydraulic pressure differential. This approach allows the electric motors to "bridge the gap" between the discrete levels offered by the rail pairings, enabling smooth and responsive

operation.

Importantly, these M/Gs are sized to handle only about 15% of the peak power demand for each degree of freedom. This low-power sizing is made feasible because the DDP manages the bulk of the energy-intensive pressure generation, allowing the smaller motors to focus on high-bandwidth modulation and energy recovery tasks. This architectural separation significantly enhances overall system efficiency while maintaining dynamic responsiveness.

Figure 2.1 provides a visual overview of this system, including the distribution of pressure rails, valve manifold logic, electric motors, and hydraulic circuits. The diagram illustrates how the various components interact to achieve efficient and coordinated control across all propulsion and work functions.

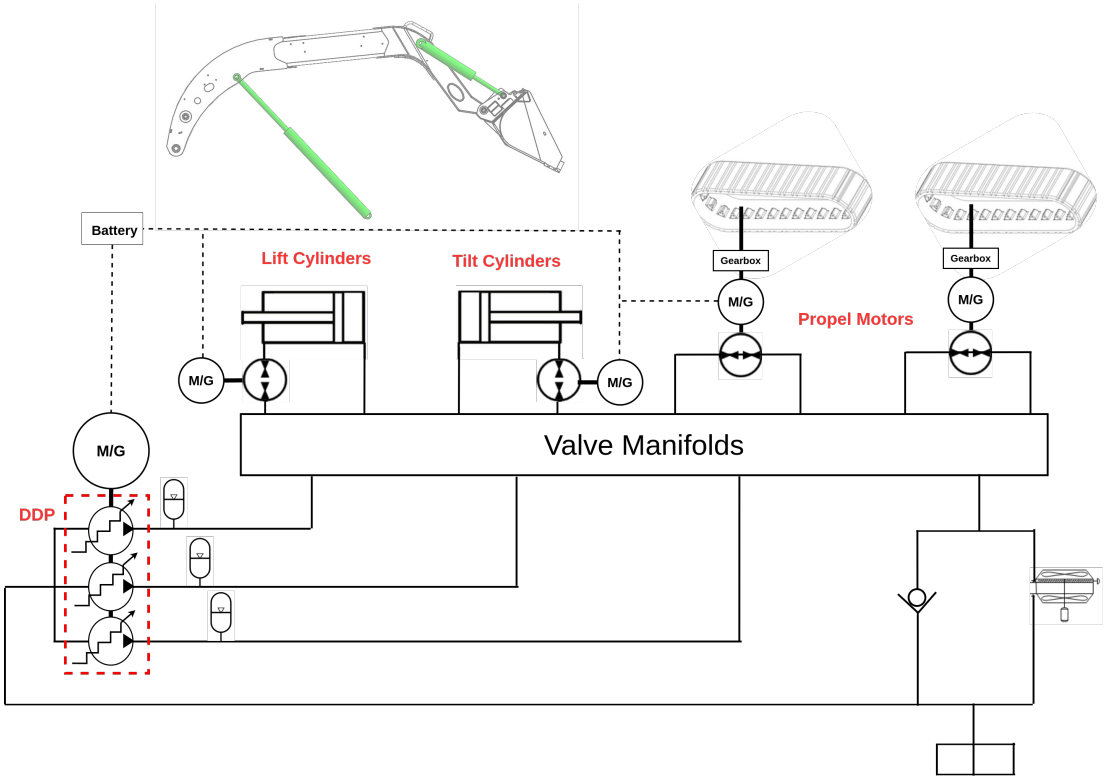

Figure 2.1: Hydraulic Architecture of Electrified HHEA Compact Track Loader

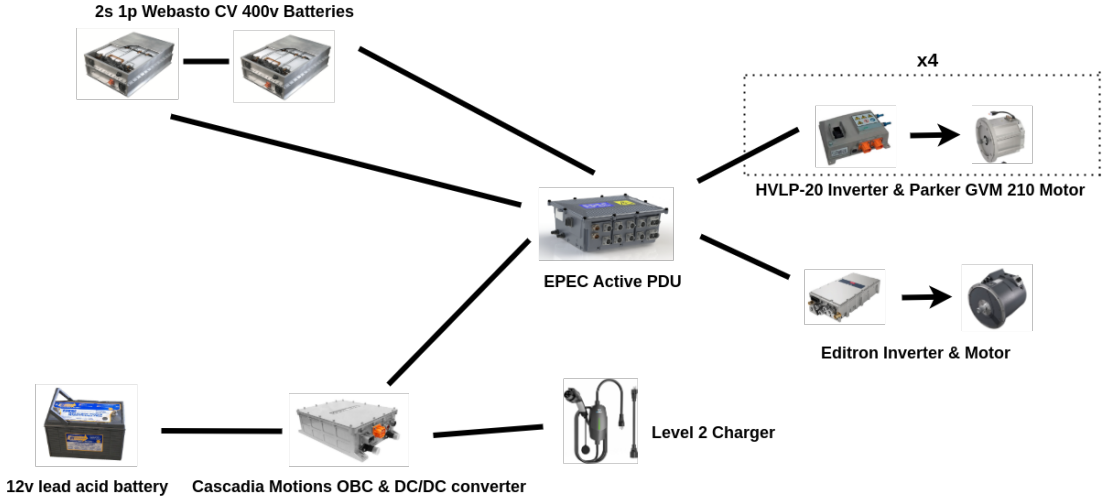

Figure 2.2: Electric Architecture of Electrified HHEA Compact Track Loader

Figure 2.2 illustrates the high-power electrical architecture of the electrified compact track loader. Power is supplied by two 400V Webasto battery packs connected in series to form an 800V DC high-voltage bus. This bus feeds into the Electric Power Distribution Unit (PDU), developed by EPEC, which serves as the central node for safely routing power to downstream high-voltage components. The PDU manages the activation of high-voltage contactors and distributes power in parallel to five independent inverters, each responsible for driving an electric motor dedicated to lift, tilt, propel, or hydraulic pump operation. These inverters convert DC to AC and allow precise motor control based on real-time system demands.

The PDU operates under commands from the Vehicle Control Unit (VCU), which serves as the central logic controller. The VCU coordinates all major subsystems, including the inverters, PDU, On-Board Charger (OBC), and other safety-critical devices. It also receives feedback from the PDU, such as contactor states, internal temperatures, and insulation monitoring status.

Battery monitoring is handled by a Vehicle Interface Gateway (VIG), which collects telemetry data—such as voltage, current, temperature, and state of charge (SOC)—and transmits it over the Battery CAN bus. This information is vital for power budgeting,

thermal management, and fault detection.

AC charging operations are managed by a Cascadia Motion On-Board Charger (OBC), which adjusts charging parameters such as voltage and current in real time based on constraints received from the Battery Management System (BMS). The BMS functions as an independent ECU, continuously monitoring battery health, balancing cells, and issuing fault warnings as needed.

The system uses two primary CAN buses: the Battery CAN bus and the Vehicle CAN bus. The VCU is the central node on the Vehicle CAN bus, issuing propel and hydraulic commands to the inverters, enabling or disabling subsystems based on system state, and monitoring feedback from all connected ECUs. In this distributed control environment, reliable and real-time CAN communication is essential for synchronizing multiple motor systems and maintaining system-level functional safety.

Figure 2.3 illustrates the overall communication topology of the vehicle, highlighting how major electronic control units (ECUs) are distributed across two CAN networks: the Battery CAN and the Vehicle CAN. It shows how the VCU, inverters, PDU, BMS, OBC, and other subsystems are logically connected within the system architecture. While the figure does not depict detailed message flow or signal-level logic, it provides a clear overview of the system's physical communication structure and emphasizes the importance of consistent and well-structured CAN integration across all nodes.

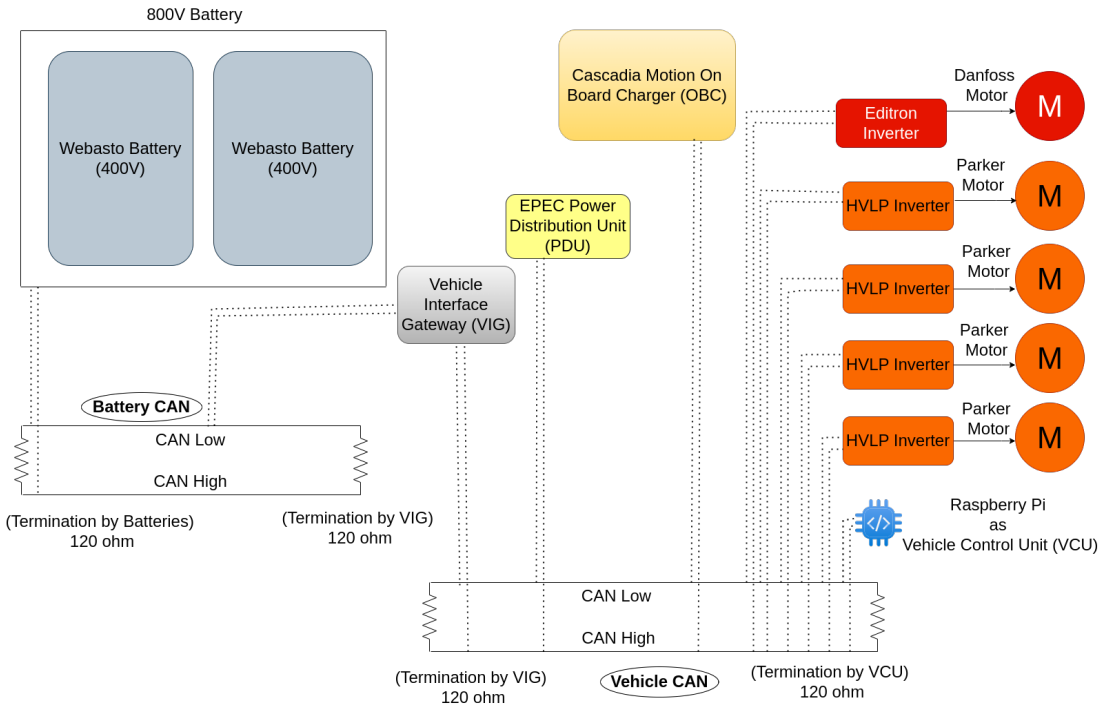

Figure 2.3: CAN Communication Architecture of the Electrified Vehicle

Together, the hybrid hydraulic-electric architecture (HHEA) and the distributed communication network form the foundation of the HHEA compact track loader. Coordinating the five electric motors, multiple ECUs, and safety-critical contactors requires a reliable software framework capable of abstracting hardware-specific behavior while offering real-time monitoring, diagnostics, and control. The framework developed in this thesis is designed to meet these exact needs, enabling scalable implementation, hardware-in-the-loop testing, and intuitive visualization across similar electrified off-road platforms.

2.3 Communication Between Components

The HHEA compact track loader relies on continuous and synchronized communication between several distributed ECUs to ensure safe and efficient operation. These components exchange messages over the SAE J1939 CAN protocol across two networks: Battery CAN and Vehicle CAN. The communication framework supports a wide variety of scenarios, including system startup, drive enable, torque command execution, charging, and fault recovery. This section describes how each ECU interacts with others in these dynamic contexts.

The Battery ECU, implemented through the Vehicle Interface Gateway (VIG), plays a central role in monitoring battery status and enabling high-voltage operation. During startup, the VIG awaits a high-voltage enable command from the VCU, along with confirmation that the vehicle is in a safe state (e.g., no crash signal, no isolation fault). It also receives ambient temperature data for thermal protection, vehicle speed for estimating dynamic load conditions, and time synchronization data to ensure accurate logging and fault analysis. The VIG continuously monitors isolation resistance to detect potential ground faults or insulation failures. If isolation resistance drops below a safe threshold—especially during pre-charge or drive enable—it may block contactor closure and send a fault warning to the VCU. Upon successful precharge and enablement, the VIG broadcasts real-time battery metrics including State of Charge (SOC), State of Health (SOH), pack voltage, current, and available energy. These values directly influence how the charger, inverter, and control logic operate under normal and peak-load conditions.

The Cascadia Motion On-Board Charger (OBC) includes both an AC charging unit and a DC-DC converter. While the AC charging function is primarily active during parked or idle states when the vehicle is docked for charging, the DC-DC converter remains active even when the vehicle is not docked. It continuously supplies 12V power to critical low-voltage systems from the high-voltage battery. The BMS, via the VCU, communicates safe voltage and current limits to the OBC based on battery thermal and electrical status. The OBC dynamically adjusts its charging profile to honor these constraints, ensuring long-term battery health and safe operation. During charging, the OBC periodically sends its output voltage, charging current, internal temperature, and

operational flags to the VCU. These flags indicate whether the charger is idle, active, or in a fault state. For example, if an over-temperature fault is triggered, the OBC sends a fault code and halts charging. The VCU responds by logging the event and updating the dashboard to alert the operator. Additionally, if the battery reaches maximum SOC or the BMS imposes stricter limits mid-charge, the VCU updates the OBC's configuration in real time to taper current or stop charging gracefully.

The EPEC Power Distribution Unit (PDU) governs the connection and disconnection of various high-voltage lines, acting as a physical gatekeeper for energy routing. During system startup, the VCU first verifies that all preconditions are met—including battery availability, VIG confirmation, and system isolation checks—before issuing a command to close the main contactors. The PDU then attempts to engage the contactors and returns confirmation messages. If a contactor fails to close or detect its own engagement, the PDU broadcasts a fault status, preventing further system progression. During runtime, the PDU continuously reports temperature, current, and voltage across its channels. In high-power scenarios—such as climbing a slope or lifting a load while driving—the VCU monitors these values to avoid thermal overload and may proactively reduce torque commands to inverters. In fault scenarios such as crash detection or inverter overcurrent, the VCU commands the PDU to open all contactors, immediately isolating the high-voltage system.

The Editron Inverter controls the motor that drives the Digital Displacement Pump (DDP), a critical component that generates the hydraulic flow supplied to the common pressure rail, enabling coordinated operation of lift, tilt, and auxiliary actuators. During initialization, the inverter undergoes self-diagnostics and reports its readiness via CAN. Once the system enters operational mode, the VCU sends speed commands to the Editron Inverter based on the flow and power demands from the common pressure rail, ensuring the DDP delivers the required hydraulic output for vehicle actuation. For example, during lifting or tilting operations, the VCU may command a constant nominal motor speed to maintain steady hydraulic flow. However, if flow demand is low or the power consumption approaches system limits, the VCU can reduce the speed to allow the DDP to operate at higher displacements or to stay within safe power boundaries, thereby optimizing efficiency and avoiding pressure surges. The inverter responds with detailed feedback including actual motor speed, rotor position (via resolver), DC link

voltage, and thermal status. If the inverter detects internal faults (such as overcurrent, undervoltage, or encoder errors), it broadcasts fault codes immediately, allowing the VCU to disable the pump and safely transition to a fault state.

The four HVLP inverters control the electric motors responsible for left and right track propulsion, as well as the two motors that drive the hydraulic actuators for lift and tilt functions. These inverters are crucial during drive enable, coordinated motion, and closed-loop control. At drive enable, the VCU sends motor activation requests along with torque and current limits tailored to battery conditions and vehicle state. For instance, if SOC is low or battery current limits are reduced due to temperature, the VCU derates torque commands across all inverters to balance power demand. During driving or work operations, the HVLP inverters send feedback on actual torque output, motor speed, position (from encoder data), and internal temperatures. This data is used by the VCU to monitor load symmetry, detect slippage or load spikes, and dynamically adjust commands. If one traction inverter experiences a thermal warning, the VCU can reduce only that side's torque to maintain drivability without a full system shutdown. During coordinated maneuvers—like lifting while driving forward—the VCU must balance torque among all inverters and prioritize commands based on operator intent and system constraints.

Across all these components, the system operates in a tightly integrated communication loop. Fault propagation is handled through prioritized message IDs, while heartbeat and watchdog signals ensure node liveness. Time-synchronized logs are critical for debugging and safety analysis, especially during rapid transients or fault transitions. The use of J1939 ensures that all components—regardless of vendor or internal implementation—can communicate using a standardized PGN/SPN structure. This approach supports scalability, modularity, and long-term maintainability of the control architecture.

In summary, each component's behavior adapts to changing system conditions, and communication is not static but responsive, fault-tolerant, and safety-critical. These dynamic interactions form the backbone of the framework developed in this thesis, ensuring that electrified off-road systems can be controlled, monitored, and debugged effectively under a wide range of operating scenarios.

2.4 Literature Review

The existing literature and tools for electrified vehicle communication frameworks, particularly those involving Controller Area Network (CAN) protocols and visualizations, illustrate diverse capabilities but also reveal notable limitations.

Patil et al. investigated an infotainment system utilizing the CAN protocol and System-on-Module (SoM) with a Qt-based application tailored for Formula-style electric vehicles [2]. Their design focused on real-time monitoring of critical parameters such as voltage, current, motor temperature, and speed. However, the system lacked integrated trace analysis, comprehensive documentation, streamlined ECU setup, embedded performance optimization, and rigorous SAE J1939 compliance. Additionally, the transparency of vehicle source code and support for plug-and-play adaptability to off-road electrified vehicles was not addressed, limiting its broader industrial applicability.

Benedetti et al. developed a digital dashboard on a low-cost embedded Raspberry Pi platform, demonstrating a Qt-based interface for displaying CAN messages via the OBD-II interface in a fully electric vehicle [3]. Although cost-effective and adaptable, this approach was constrained by reliance on OBD-II rather than the more robust SAE J1939 standard, and it lacked advanced features like extensive trace analysis tools, ease of new ECU integration, optimized embedded C++ frameworks, and comprehensive documentation, which are critical in off-road electrified vehicle environments.

Commercial solutions such as Vector CANoe and EPEC plug-and-play displays offer powerful features including advanced CAN network simulation, diagnostic protocol support, graphical dashboard creation, and rugged HMI integration for real-time vehicle monitoring [4, 5]. However, their practical adoption in academic or specialized electrified off-road vehicle projects is often hindered by proprietary restrictions, significant licensing costs, lack of open-source adaptability, absence of full transparency into vehicle code, and difficulties in rapid ECU integration.

MATLAB and Simulink offer sophisticated capabilities for electric vehicle development, simulation, and prototyping but suffer from high licensing costs, complex workflows, proprietary constraints, and limited native support for open-source flexibility and rapid ECU integration [6].

Open-source alternatives such as SocketCAN provide fundamental CAN communication infrastructure but lack user-friendly graphical interfaces, extensive trace analysis, detailed documentation, and robust visualization capabilities [7].

The Open-SAE-J1939 project, despite offering standardized J1939 protocol implementations in C, does not include advanced visualization, infotainment integration, or comprehensive trace analysis capabilities, limiting its direct application to advanced embedded systems [8].

GENIVI/COVESA Open Infotainment Stack and Automotive Grade Linux (AGL) are substantial open-source infotainment frameworks but lack detailed trace analysis tools, optimization for embedded performance through C++ Qt frameworks, seamless integration with additional ECUs, and full compliance with SAE J1939 standards, reducing their effectiveness in specialized off-road electrified vehicle applications [9, 10].

Eclipse Kuksa supports connected vehicle functionalities but similarly lacks integrated CAN trace analysis, optimized Qt-based embedded performance, ease of ECU integration, rigorous J1939 protocol adherence, and extensive documentation, restricting its practical use in robust off-road electrified vehicles [11].

The Python-based can-j1939 library simplifies J1939 integration but does not provide comprehensive infotainment capabilities, advanced real-time visualization, embedded system optimization, or extensive analytical toolsets, hindering its practical usability in complex electrified vehicle systems [12].

Lastly, MATLAB's Instrument Cluster Toolbox, while effective for dashboard prototyping, is not designed for embedded deployment and lacks features such as integrated trace analysis, streamlined ECU integration, optimized runtime performance, and full adherence to the J1939 standard—necessitating substantial additional development to arrive at a production-ready solution [13].

To address these comprehensive gaps identified in the literature and existing tools, the proposed thesis framework integrates advanced infotainment functionalities, detailed trace analysis capabilities, simplified and rapid setup for new ECU integration with Vehicle Control Units (VCUs), an optimized embedded C++ Qt-based software framework, strict compliance with SAE J1939 protocol standards, complete transparency of the vehicle source code and DBC file integration[1] , detailed documentation, and robust

[1] See Appendix A for an overview of DBC structure and decoding/encoding process.

analytic tools within a fully open-source and plug-and-play architecture. This unified approach effectively caters specifically to the complex requirements of electrified off-road vehicle applications.

2.5 Chapter Summary

This chapter described the hybrid hydraulic-electric vehicle developed at the University of Minnesota, outlined the critical role of communication protocols like J1939 in coordinating its subsystems, and reviewed the current state of tools, visualization, and testing frameworks. It highlighted a research gap in software standardization for communication, which this thesis addresses. The next chapter presents the architecture, design, and implementation of the proposed software framework.

Chapter 3

System Architecture and Framework Design

3.1 Chapter Overview

This chapter outlines the system architecture and embedded software framework developed to implement J1939 CAN communication for an electrified off-road vehicle. The architecture integrates both hardware and software layers, with the Vehicle Control Unit (VCU) acting as the central controller for communication across subsystems including the high-voltage battery, inverters, On-Board Charger (OBC), Power Distribution Unit (PDU), and the Vehicle Interface Gateway (VIG).

The software framework is built on a modular, Qt-based embedded platform designed to manage CAN messaging, decode and encode signals using DBC (Database CAN) files, and enforce timing for safety-critical communication. DBC files define the structure of CAN messages, specifying signal positions, lengths, scaling factors, and units, enabling consistent interpretation of raw data. The framework also features a JSON-driven, condition-based FSM (Finite State Machine) to support flexible control logic for tasks such as startup, shutdown, charging, and fault handling.

This chapter describes the VCU's internal architecture, task coordination, data flow between software layers, and the runtime logic for transmitting and receiving CAN messages. It also covers the design choices for signal synchronization, system abstraction, and safety message generation using SHM and SDM structures. Together, these

elements form the foundation for the real-time dashboard interface and serve as the communication backbone for vehicle testing and validation.

3.2 Data Transmit and Receive Framework

The Data Transmit and Receive Framework plays a vital role in the overall CAN communication system of the Vehicle Control Unit (VCU). It handles the bidirectional exchange of CAN messages through a tightly integrated yet modular design, ensuring that the latest vehicle data is transmitted and received with reliability and determinism. The internal logic revolves around three central components: the `CanMessageSender`, the `CanMessageReceiver` (specifically its `processFrames()` function), and a shared memory structure called `dataVector`. These modules interact in a defined sequence, shown in Figure 3.1, to maintain a consistent and up-to-date representation of the vehicle's operational state.

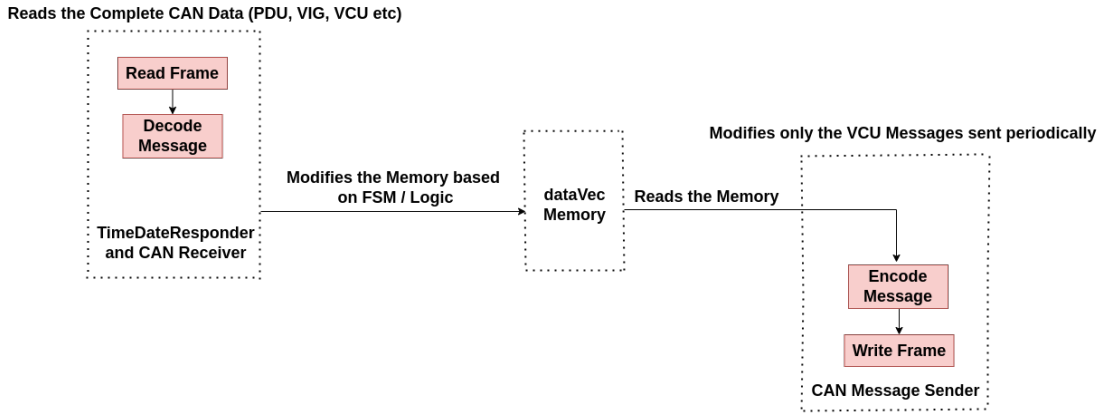

Figure 3.1: CAN Transmit and Receive Framework

The communication cycle begins with the creation of the `dataVector` memory. This structure serves as the central repository for all decoded CAN signals and associated status flags. It is initialized during system startup and is accessible throughout the VCU's runtime by all logic modules. Once initialized, the framework sets up the CAN receive and transmit infrastructure using two core classes—`TimeDateResponder` (which acts as

the CAN frame receiver) and `CanMessageSender` (responsible for CAN transmission).

When a new CAN frame is received on the vehicle's CAN bus, it is passed to the `TimeDateResponder`, which handles incoming frames in a dedicated receive thread. The raw payload is decoded using signal definitions derived from the DBC file. The resulting signal values are written directly into the `dataVector` memory, replacing the previous values and updating the internal representation of the system state. This ensures that the shared memory always reflects the most recent vehicle conditions.

Parallel to this, the `CanMessageSender` continuously runs in a separate transmit thread. At every iteration of its cycle, it evaluates which messages are due for transmission based on their predefined cycle times. For each message that is ready to be sent, the system first checks whether it is linked to a signal in the `dataVector`. If it is, the message's signal fields are dynamically updated using the latest values stored in memory. This guarantees that outgoing messages always contain the freshest available data.

After updating the signal fields, the message undergoes signal encoding, converting structured values into the raw byte format defined by the J1939 or custom CAN specification. The encoded message is then written to the CAN interface, completing one full transmission cycle.

This mechanism is designed to be both reactive and deterministic. By decoupling the signal acquisition (receive) from the message generation (transmit), the architecture ensures that no delay or jitter propagates from one component to another. It also enables seamless scaling—new messages or signal definitions can be introduced simply by extending the `dataVector` or adjusting the DBC file, without changing the core logic.

For example, to store a signal such as the battery pack's maximum voltage, an entry is inserted into the `dataVector` as shown in Algorithm 1. Here, the PGN (Parameter Group Number), signal name, and signal value are assigned to a specific index within the vector.

Algorithm 1: Set PGN, Signal Name, and Value for Index 3 in `dataVec`

Input : A vector `dataVec` of CAN signal structures

Output: Updated entry at index 3 with PGN, signal name, and value

1 `dataVec`[3].pgn ← 0x1806E5F4

2 `dataVec`[3].signal ← "BMS_Max_Voltage"

3 `dataVec`[3].value ← 300

This shared memory framework simplifies data flow across the system, provides a single source of truth for all modules, and ensures a synchronized exchange of information between the vehicle and the controller. Through this design, the VCU maintains consistent behavior, real-time responsiveness, and high modularity for future extensions or diagnostics.

The `dataVector` thus serves not only as a storage mechanism but as the central system state within the VCU architecture. Incoming CAN frames update this shared memory with the latest vehicle conditions, while all outgoing messages, FSM transitions, and dashboard displays derive their behavior from the most current values stored in this state. This memory-centric approach abstracts the complexities of CAN communication and allows upper-level application logic to interact with a clean and consistent data interface. The following section builds on this foundation by exploring how this principle is embedded within the larger software architecture of the VCU, detailing how tasks are scheduled, threads are isolated, and control logic is structured around this shared state.

3.3 VCU Software Architecture

This section describes the layered architecture of the Vehicle Control Unit (VCU) developed for an electrified off-road vehicle. The software system is implemented using a modular, Qt-based embedded framework and interfaces with various vehicle subsystems over a CAN bus network. The architecture is illustrated in Figure 3.2.

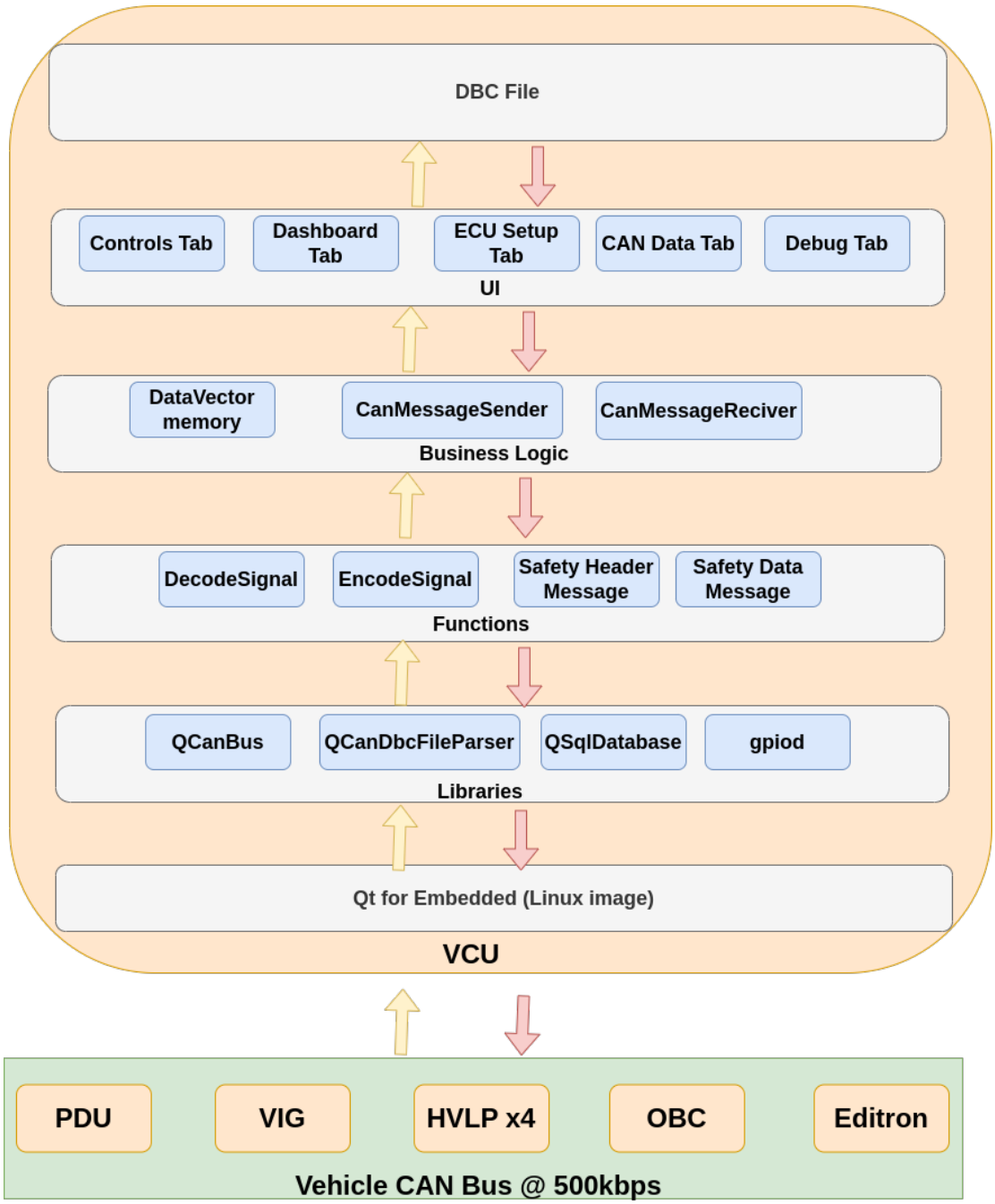

Figure 3.2: VCU Architecture

3.3.1 Vehicle CAN Network Layer

At the foundation of the system lies the **Vehicle CAN Bus**, operating at 500 kbps. It connects essential vehicle subsystems such as the Power Distribution Unit (PDU), Vehicle Interface Gateway (VIG), four High Voltage Logic Power modules (HVLP x4), Onboard Charger (OBC), and the Editron inverter. These hardware components communicate using J1939 messages transmitted over the CAN network. The VCU serves as the central coordinator, interfacing with each module to monitor state, exchange commands, and manage system-level behavior.

3.3.2 Embedded Runtime and Libraries

The VCU software runs on a Linux-based embedded platform, built with **Qt for Embedded Linux**. This environment supports UI rendering and backend processing. At its core, the software uses several key libraries:

- `QCanBus`: Abstracts CAN interface and enables message transmission and reception.
- `QCanDbcFileParser`: Parses DBC files to extract message and signal structure.
- `QSqlDatabase`: Manages configuration data and optional logging functionality.
- `gpiod`: Interfaces with GPIOs for low-level control or safety IO requirements.

These libraries form the backbone of communication and hardware abstraction, enabling the upper layers to remain modular and platform-independent.

3.3.3 Functional Modules

Above the library layer are the core **functional modules** responsible for processing CAN messages. Functions like `EncodeSignal` and `DecodeSignal` convert between raw byte arrays and physical signal values. Safety-related messages are handled separately through dedicated modules like `Safety Header Message` and `Safety Data Message`,

ensuring that safety-critical data adheres to the expected format and timing requirements[1] . These functions are tightly integrated with the DBC parser and CAN interface libraries.

3.3.4 Business Logic Layer

The **business logic layer** manages runtime behavior of the vehicle's CAN communication. The `CanMessageSender` and `CanMessageReceiver` modules handle transmission and reception of J1939 messages in periodic and event-driven modes. A centralized `DataVector memory` structure maintains the latest values of all relevant signals, acting as a thread-safe abstraction between raw CAN data and higher-level logic or UI interactions.

3.3.5 User Interface Layer

The topmost layer is the **User Interface (UI)**, which enables users to interact with the system in real-time. It consists of several modular tabs:

- **Controls Tab**: Sends commands to actuators and vehicle subsystems.
- **Dashboard Tab**: Displays live vehicle states and performance metrics.
- **ECU Setup Tab**: Allows configuration of electronic control units.
- **CAN Data Tab**: Visualizes raw and decoded CAN traffic.
- **Debug Tab**: Provides system status, fault diagnostics, and developer tools.

The UI is tightly coupled with the `DataVector`, enabling real-time visualization and interaction with signals processed by the business logic.

3.3.6 DBC Integration and Modularity

At the highest abstraction, the **DBC File** acts as the source of truth for message structures, signal encoding rules, and communication definitions. While each component defines the messages it transmits or expects through its own DBC specification, these

[1] For a detailed explanation of Safety Header and Safety Data Messages, see Appendix A.7

are merged into a single combined DBC file that the system uses at runtime. This unified file is parsed using the `QCanDbcFileParser` and leveraged across all layers of the stack—from encoding/decoding logic to UI display formatting. This design decouples the implementation from hardcoded message formats and enables easy migration to other platforms by simply replacing the combined DBC file.

3.4 Signal-Slot Communication Mechanism

The application relies heavily on Qt's signal-slot mechanism to facilitate asynchronous communication between components. This design promotes modularity and clean separation between message acquisition, processing, and data management. Figure 3.3 illustrates the interaction between different classes involved in this mechanism.

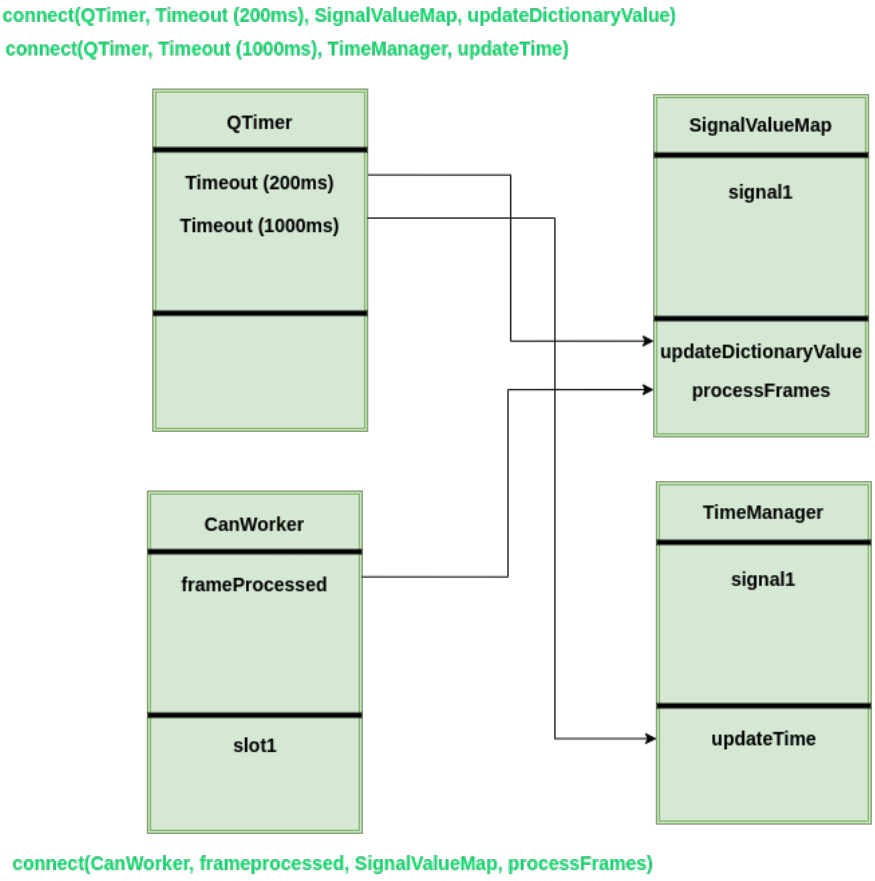

Figure 3.3: Signal-Slot Mechanism

3.4.1 Periodic Signal Updates Using QTimer

The `QTimer` class is used to trigger periodic actions within the application. Two separate timers are instantiated: one with a timeout interval of 200 ms and another at 1000 ms. These timers are connected to relevant processing slots using Qt's `connect` function. Specifically, the 200 ms timer is connected to the `SignalValueMap::updateDictionaryValue()` slot, which is responsible for refreshing the signal values maintained in a central dictionary structure. Meanwhile, the 1000 ms timer updates the system timestamp by invoking `TimeManager::updateTime()`.

3.4.2 CAN Frame Handling with CanWorker

Incoming CAN frames are handled by the `CanWorker` class. Once a frame is received and processed internally, the `frameProcessed` signal is emitted. This signal is connected to the `processFrames()` slot of the `SignalValueMap` class. The connection enables decoupling of frame acquisition from decoding and interpretation, allowing signal processing to be modular and scalable. Additionally, `CanWorker` contains internal slots like `slot1` for specialized frame-specific operations.

3.4.3 Centralized Signal and Time Management

The `SignalValueMap` class acts as the central data structure for storing and updating decoded signal values. It exposes two public slots: `updateDictionaryValue()` for time-triggered updates, and `processFrames()` for event-driven updates upon frame reception. These slots ensure the signal values reflect both time-based changes and new incoming CAN data. Similarly, the `TimeManager` class maintains global time synchronization and receives periodic triggers via `updateTime()`, which ensures that time-based logic across the application remains consistent.

3.4.4 Slot Connections

The entire mechanism is held together by three primary signal-slot connections:

- `connect(QTimer, Timeout (200ms), SignalValueMap, updateDictionaryValue)`
- `connect(QTimer, Timeout (1000ms), TimeManager, updateTime)`
- `connect(CanWorker, frameProcessed, SignalValueMap, processFrames)`

These connections reflect a clean and reactive architecture where timers and worker threads push data into a centralized repository, keeping the UI and control layers responsive and updated with the most recent information.

3.5 VCU Execution Flow

The VCU software follows a structured execution flow from initialization to runtime operation and shutdown. The core logic is organized around modular components that interact via message structures, timers, and CAN signals. Figure 3.4 presents an overview of this flow.

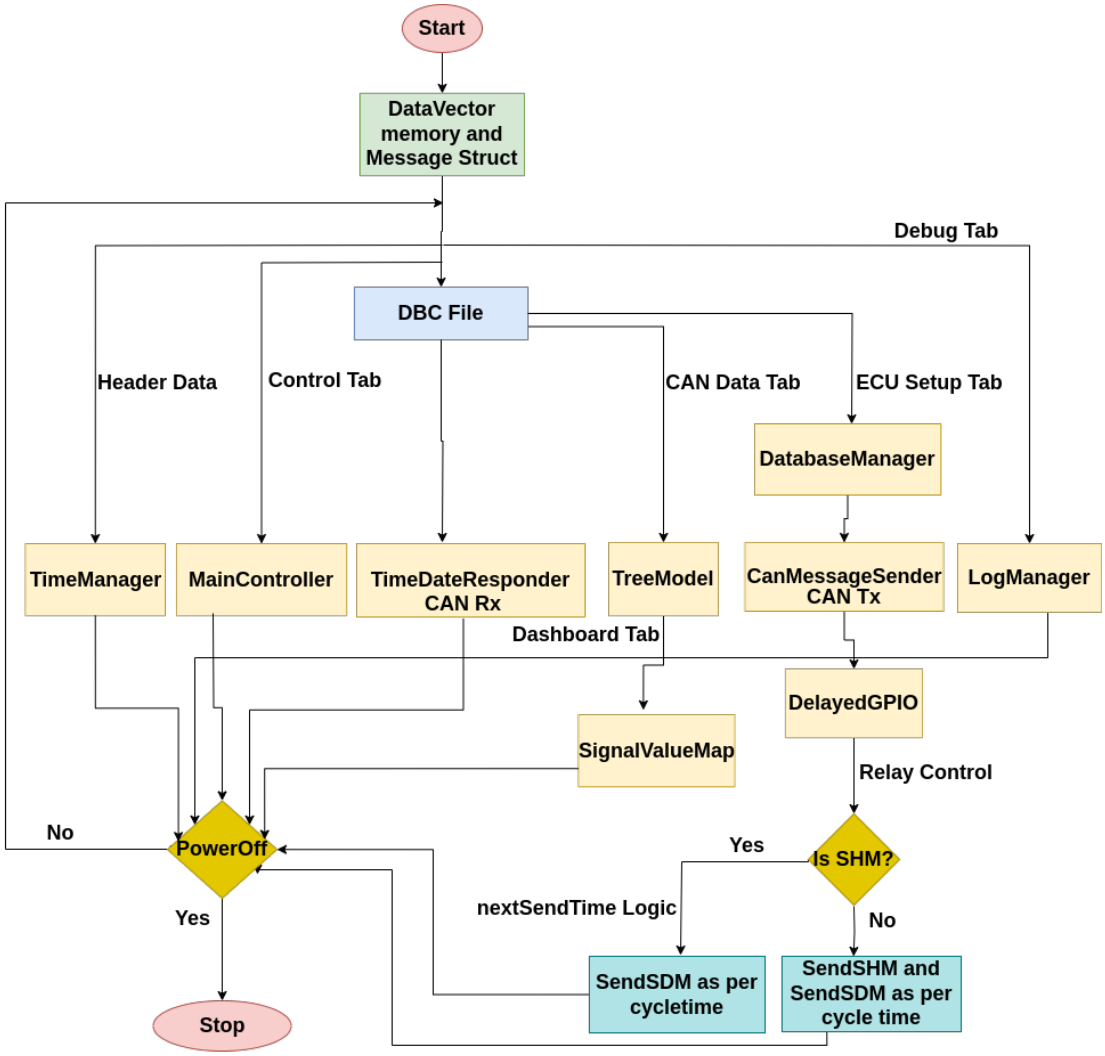

Figure 3.4: VCU Software Flowchart

3.5.1 Initialization Phase

The system begins with the **Start** block, triggering initialization of key structures such as the `DataVector memory` and CAN message structures. This memory abstraction acts as the backbone of inter-module communication, storing all signal states in a thread-safe manner. Next, the **DBC File** is loaded and parsed to extract signal definitions, frame formats, and scaling rules used throughout the application.

3.5.2 Component Instantiation and Setup

After parsing the DBC file, the system generates internal data structures representing the CAN messages and signals defined in the network. This includes message identifiers (e.g., PGNs and source addresses in J1939), signal names, byte offsets, bit lengths, scaling factors, units, and value ranges. These parsed structures are used to dynamically encode outgoing signals and decode incoming raw CAN frames in real time. For each received CAN message, its identifier is matched against the DBC-derived message definitions, allowing the system to extract and update corresponding signal values. This parsing step enables flexible handling of protocol changes and simplifies integration with new ECUs by eliminating the need for hardcoded message formats.

Building upon this parsed information, the system proceeds to instantiate multiple core components are instantiated and configured. These include:

- `TimeManager` – maintains and synchronizes global system time.
- `MainController` – orchestrates vehicle-level logic and high-level state transitions.
- `TimeDateResponder (CAN Rx)` – responds to external time sync messages.
- `TreeModel` – organizes signal values in a hierarchical structure for UI display.
- `DatabaseManager` – manages configuration and logging database access.
- `CanMessageSender (CAN Tx)` – sends CAN frames periodically.
- `LogManager` – records system activity and faults.
- `DelayedGPIO` – manages delayed actuation logic based on timers or conditions.

`SignalValueMap` is also instantiated and acts as the central signal repository. It interacts with all runtime logic including CAN message processing, GUI updates, and safety checks.

3.5.3 Conditional Flow and State Handling

During runtime, two main decision blocks control the flow. First, a check for **PowerOff** determines whether the system should shut down. If power-off conditions are met, the system transitions to the **Stop** block, halting all operations gracefully.

Another critical decision point is the **Is SHM?** (Safety Header Message) condition. Based on this check, the system sends only the SDM (Safety Data Message) or both SHM (Safety Header Message) and SDM according to their defined cycle times. This logic ensures that safety-critical messages adhere to communication timing and protocol constraints.

3.5.4 Runtime Operation and Monitoring

Once all components are active, the system continuously monitors signal values, handles CAN Rx/Tx, and logs activity. `CanMessageSender` operates cyclically, pushing out vehicle state or control commands. Simultaneously, `SignalValueMap` is updated with newly received frames and shared with other components, including UI, safety logic, and data loggers.

3.5.5 Shutdown Sequence

If a power-off condition is detected (due to shutdown command, safety fault, or external signal), the VCU stops all real-time tasks and enters the **Stop** state. This ensures that CAN communication is halted, GPIOs are set to safe states, and all log data is safely stored.

3.6 Timed Message Transmission Logic

To ensure timely and deterministic communication over the CAN bus, safety-critical messages must be transmitted at predefined cycle times. Algorithm 2 illustrates a

time-based scheduling mechanism used to send messages defined in the Safety Header Message (SHM) list. Each message contains its own unique cycle time, which governs the interval at which it must be transmitted.

Algorithm 2: Timed Message Sending Based on Cycle Time

Input : A list of messages with their cycle times in SHM

Output: Messages sent at their respective cycle times

Initialization:
foreach $m \in$ *messages* **do**
  if m.*SHM is not empty* **then**
    `nextSendTime`$[m.\texttt{id}] \leftarrow$ current time

Runtime Loop:
$now \leftarrow$ current time
foreach $m \in$ *messages* **do**
  if $now \geq$ *nextSendTime*$[m.id]$ **then**
    `SendMessage`(m)
    `nextSendTime`$[m.\texttt{id}] \leftarrow$ `nextSendTime`$[m.\texttt{id}] + m.\texttt{SHM}[0].\texttt{cycleTime}$

The algorithm is divided into two phases: initialization and runtime operation. During initialization, the next scheduled send time for each message is set to the current system time, assuming it contains an SHM entry. During the runtime loop, the system continuously checks whether each message has reached its scheduled send time. If so, the message is sent using the `SendMessage` function, and its next send time is updated by adding the defined cycle time. This ensures messages are sent at precise intervals without drift.

While this logic is centered around SHM scheduling, the same timing control applies to the corresponding Safety Data Messages (SDMs), which must follow the SHM within a defined time delta to maintain J1939-76 compliance. The framework ensures both messages are synchronized and transmitted deterministically within their required cycles. This mechanism integrates seamlessly into the main control loop of the VCU, providing predictable message delivery for critical vehicle functions.

3.7 Qt for Embedded Framework

The development of the Vehicle Control Unit (VCU) dashboard and J1939 CAN communication logic is based on **Qt for Embedded**, specifically using the Qt for Automotive stack. This embedded framework is tailored for running multi-threaded applications on resource-constrained platforms and integrates key libraries that simplify the handling of CAN communication, user interface rendering, and hardware abstraction.

As illustrated in Figure 3.5, the architecture includes a CAN network connecting multiple electronic subsystems: a high-voltage battery pack (800V), the Vehicle Integration Gateway (VIG), a Raspberry Pi-based VCU, and an On-Board Charger (OBC). Each of these is implemented as a CAN node with its own Electronic Control Unit (ECU). The vehicle CAN bus operates at a baud rate of 500 kbps, with the VIG following the J1939-76 application layer protocol. According to this protocol, the VCU (Raspberry Pi) must first transmit a **Safety Header Message (SHM)**, which includes the message number and checksum. This is then followed by a **Safety Data Message (SDM)** that carries time-critical parameters such as cruise control commands, voltage limits, and high-voltage status.

Once the SHM and SDM are sent, the VIG responds with battery and cell-level data, including voltage, current, power, state of charge (SOC), and state of health (SOH). The VCU reads voltage and current limits from the VIG and forwards them to the OBC ECU. The charger responds with details like output voltage, current, internal temperature, and fault conditions. This round-trip data exchange ensures tight coordination between all powertrain components, a necessity in an electrified vehicle system.

The third block in the framework diagram represents a generalized abstraction that allows new ECUs to be added dynamically. Through the Qt-based dashboard interface, users can add new signal entries via a grid-based UI. Each new signal can be assigned a cycle time and live value, which is then used by the system to handle message encoding and CAN transmission. This user-driven expansion capability enhances the modularity and scalability of the overall architecture.

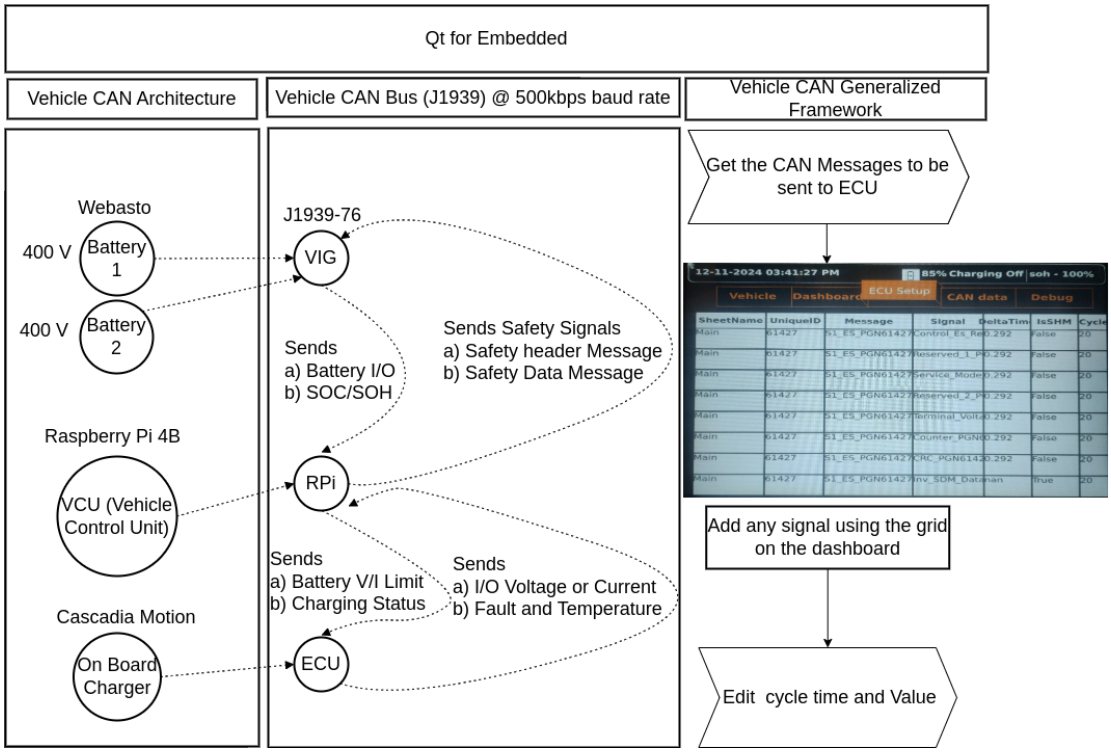

Figure 3.5: Qt Framework

3.7.1 CAN UI Framework Development Using Qt for Embedded

The CAN-based user interface, which facilitates interaction with the vehicle's control and monitoring system, is entirely developed using Qt for Embedded. This framework provides a compact, efficient runtime environment tailored for embedded Linux platforms such as the Raspberry Pi. It offers developers the flexibility to build custom graphical interfaces while tightly integrating application logic, hardware I/O, and real-time CAN communication. The entire development and deployment process is shown in Figure 3.6, which outlines the workflow from source code to on-device execution.

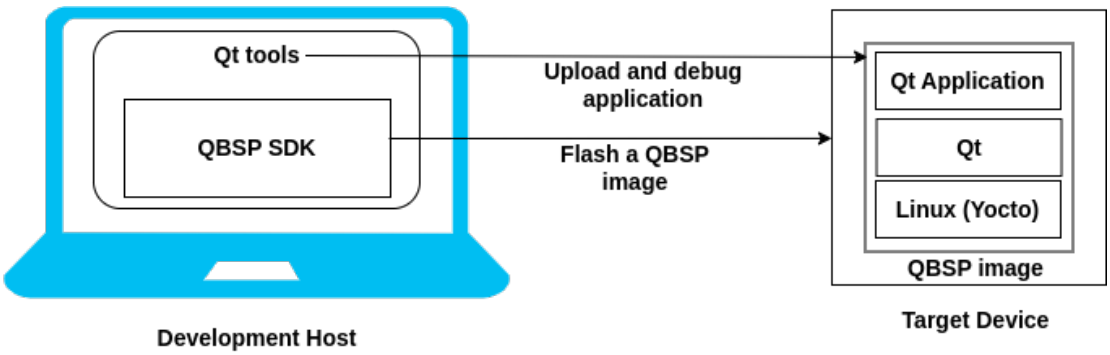

Figure 3.6: Qt for Embedded Development and Deployment Workflow

The development process begins with setting up the build environment using the Yocto Project[14], a powerful toolchain designed to create custom Linux distributions for embedded systems. The base Raspberry Pi image provided by the meta-raspberrypi layer is extended to support all required dependencies. Since the default image lacks GPIO and some specialized hardware control features, the build configuration must be manually adjusted. This involves modifying BitBake recipe files (.bb) to include libraries such as the GPIO control module and declaring their dependencies explicitly within the build system. These additions ensure that when the image is built, it includes support for both UI-level interactions and low-level hardware access.

After all necessary packages are configured, the image is compiled using BitBake. This compilation step builds the Linux kernel, device drivers, Qt libraries, and any additional modules required by the VCU application. The build process is compute-intensive and may take several hours to complete depending on the capabilities of the host machine. Once the image is successfully compiled, it is flashed onto the Raspberry Pi. The result is a bootable system image that includes the full Qt runtime, CAN bus support, GPIO handling, and any other custom functionalities integrated during the build.

During application development, Qt Creator[15] serves as the primary integrated development environment. It supports live development by enabling source-level debugging and rapid deployment over a wireless connection. User interface components are designed using Qt Designer, a visual tool for building responsive dashboards and

input controls. Meanwhile, application logic is implemented in C++ and tightly coupled with Qt's signal-slot architecture, allowing seamless communication between UI elements and backend processes. Changes to the UI or logic can be tested directly on the Raspberry Pi without requiring a full image rebuild, which significantly accelerates development and debugging.

The framework also integrates key Qt libraries for handling CAN communication. The `QCanBus` library abstracts the lower-level CAN interface and provides a consistent API for reading and writing frames. Paired with `QCANDBParser`, which parses DBC files to extract signal definitions and frame layouts, these libraries enable efficient encoding and decoding of J1939 messages within the application[16]. This allows the dashboard to display real-time decoded data from the vehicle while also letting users or control logic send commands over the CAN bus using structured messages.

Overall, this development strategy—combining Qt for Embedded[17], Yocto-based image customization, and live deployment through Qt Creator—results in a responsive, extensible, and production-ready HMI (Human-Machine Interface) tailored for off-road electrified vehicles. It simplifies complex workflows like integrating new ECUs, extending CAN signals, and updating firmware, all while maintaining a consistent and intuitive interface for vehicle operators and developers alike.

3.7.2 Thread Coordination and State-Based Logic

The VCU software architecture is based on multi-threaded execution where communication threads operate independently yet remain synchronized through shared memory and event-driven logic. Two core threads—one for transmission and one for reception—run concurrently and interact through a centralized data structure known as the `dataVector`. In addition to this, a JSON-defined finite state machine (FSM) governs the conditional logic for updating signals, interpreting frame data, and determining state transitions.

The **transmit thread**, implemented within the `CanMessageSender` module, is responsible for continuously sending CAN messages at predefined cycle times. For every message that is due to be transmitted, the sender reads the corresponding signal values from the `dataVector`. If any state-based rules are defined in the FSM configuration, the sender dynamically selects signal values or alters frame contents based on the current

system state or flags.

The **receive thread**, handled by the `CanMessageReceiver` (particularly the `processFrames()` function), listens for incoming CAN frames in real-time. Upon receiving a frame, the thread matches its CAN ID with signal definitions parsed from the DBC file. Additionally, it references the FSM JSON configuration to apply conditional logic—updating signal values only when specific criteria are met. These criteria may include value thresholds, Boolean flags, or current FSM state.

3.7.3 Example Workflow Using JSON-Based State Logic

To illustrate the integration of state transitions, consider the following scenario: a CAN frame with ID `0x18EF21F3` is received. Based on the current FSM state (e.g., `Operational`) and additional signal flags stored in the `dataVector`, the system selectively decodes and updates signals. This logic is governed by JSON-defined transitions such as:

```
{
  "state": "Operational",
  "transitions": [
    {
      "condition": {
        "frameId": "0x18EF21F3",
        "dataVec[5].value": 0
      },
      "actions": [
        {
          "signal": "VIC.PropA_C3_PGN61184.UChLimLong",
          "targetIndex": 3
        },
        {
          "signal": "VIC.PropA_C3_PGN61184.IChLimLong",
          "targetIndex": 4
        }
      ]
    }
  ]
```

```
}
```

Listing 3.1: Example State Transition Configuration in JSON

This JSON configuration instructs the system to update indices 3 and 4 of the `dataVector` only if the frame ID matches and a certain condition (e.g., `dataVec[5].value == 0`) is satisfied. The decoding logic uses the `SignalValueMap` to retrieve the corresponding signal values and commit them to memory.

Functionally, this workflow captures a critical communication task: reading the battery's **voltage and current charging limits**—specifically `UChLimLong` and `IChLimLong` from the VIG, and updating them in the centralized memory structure. These values are then forwarded to the On-Board Charger (OBC) by the VCU to configure the charging parameters in real time.

The same workflow can be expressed programmatically as shown in Algorithm 3:

Algorithm 3: Frame Decoding with State-Based Logic from `SignalValueMap`

Input : A CAN frame, vector `dataVec`, and `signalValueMap` with keys as ``node.message.signal''

Output: Updates to `dataVec[3]` and `dataVec[4]` based on FSM logic

```
if frame.frameId() = 0x18EF21F3 and dataVec[5].value = 0 then
    if ''VIC.PropA_C3_PGN61184.UChLimLong'' ∈ signalValueMap then
        dataVec[3].value ←
         signalValueMap[''VIC.PropA_C3_PGN61184.UChLimLong'']
    if ''VIC.PropA_C3_PGN61184.IChLimLong'' ∈ signalValueMap then
        dataVec[4].value ←
         signalValueMap[''VIC.PropA_C3_PGN61184.IChLimLong'']
```

Once these values are stored in memory, the `CanMessageSender` thread, upon reaching the corresponding message cycle time, reads the updated signal values from the `dataVector`, encodes them using the appropriate signal definitions, and transmits the frame. This seamless coordination between the receive and transmit threads—mediated through memory and state-driven logic—ensures that the system remains synchronized and reactive to changing vehicle conditions.

In essence, the JSON FSM enables clean separation of behavior rules from C++

code, allowing easier modification of transition logic without recompiling. It also supports scalable, event-driven updates across various CAN IDs and signal groups, while maintaining a deterministic real-time communication loop.

For a detailed example of how the FSM logic is implemented in JSON and integrated into the VCU communication workflow, refer to Appendix C.2.

3.8 Chapter Summary

In this chapter, the complete system architecture for the electrified off-road vehicle was presented, emphasizing the modular integration of hardware components and a scalable software communication framework built around the J1939 CAN protocol. The Vehicle Control Unit (VCU) was shown to be the core orchestrator, handling real-time messaging, safety logic, and subsystem coordination through a unified, DBC-based interface.

The design also incorporated a JSON-driven, condition-based finite state machine (FSM), which enabled event-based control logic for start-up, charging, and shutdown processes. This modular structure was critical for ensuring system flexibility, code reusability, and testability across both HIL and real-vehicle environments.

In the following chapter, we shift focus to the design and implementation of the human-machine interface (HMI) developed using Qt. This dashboard played a crucial role in interacting with the underlying system, enabling real-time control, status visualization, and message triggering via CAN. The chapter covers both the UI design philosophy and the backend integration with the embedded framework described here.

Chapter 4

Dashboard Design and Implementation

4.1 Chapter Overview

This chapter describes the design and implementation of a Qt-based dashboard interface developed for the electrified compact track loader. Acting as the human-machine interface (HMI), the dashboard enables real-time monitoring, control, diagnostics, and visualization of all relevant CAN signals across vehicle subsystems. The dashboard is built using the Qt framework with QML for frontend components and C++ for backend logic. Qt Creator and Qt Designer were used during development for efficient UI design, code integration, and deployment to the Raspberry Pi-based Vehicle Control Unit (VCU), which interfaces with a touchscreen display for operator interaction.

4.2 Dashboard Features and Workflow

4.2.1 Vehicle Dashboard UI Design

The dashboard is organized into five main tabs—Vehicle, Dashboard, ECU Setup, CAN Data, and Debug—each serving a distinct purpose in the vehicle's operation and diagnostics. These tabs are implemented as modular QML components dynamically loaded by the main UI controller. The modularity allows seamless switching between interfaces

and supports future extensibility for new features.

4.2.2 Control Screen

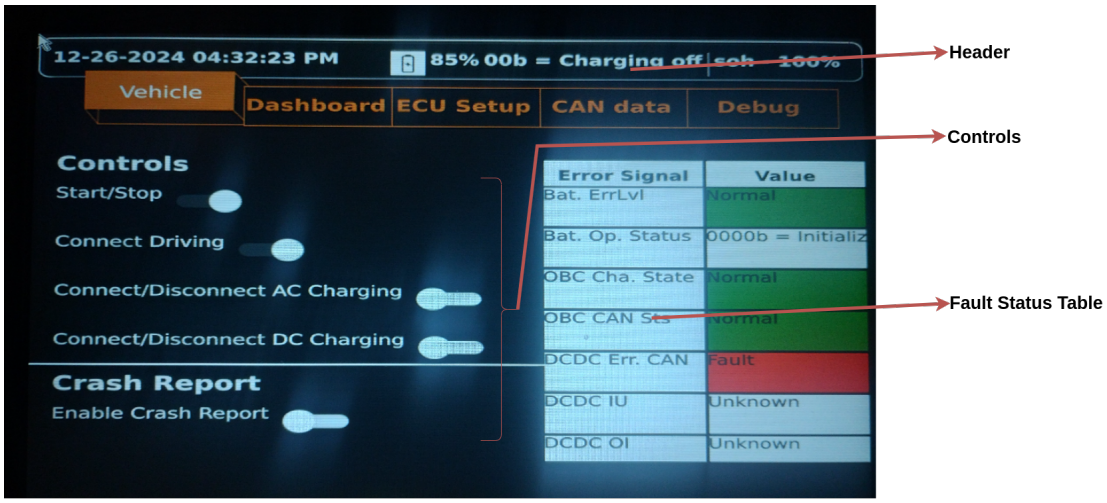

Figure 4.1: Vehicle Tab – Start/Stop and Charging Controls

The **Vehicle** tab provides key system-level controls (Figure 4.1). It allows the operator to initiate the battery start sequence, which in turn triggers periodic CAN initialization messages, such as Cruise Control and HVES Control PGNs, to keep the battery and VIG active. Additionally, toggles are provided for initiating AC and DC charging based on operational mode. These commands are translated into appropriate J1939 messages with specific signal values, ensuring that the high-voltage system behaves safely and predictably.

The interface also displays critical status indicators such as fault codes, crash status, battery state of charge (SOC), and permission-to-close conditions. These updates reflect real-time feedback from the VIG and other ECUs, helping the user verify correct sequence execution before attempting to engage drive mode or enable charging. This tab acts as the primary control center for core energy-related operations.

4.2.3 Dashboard Screen

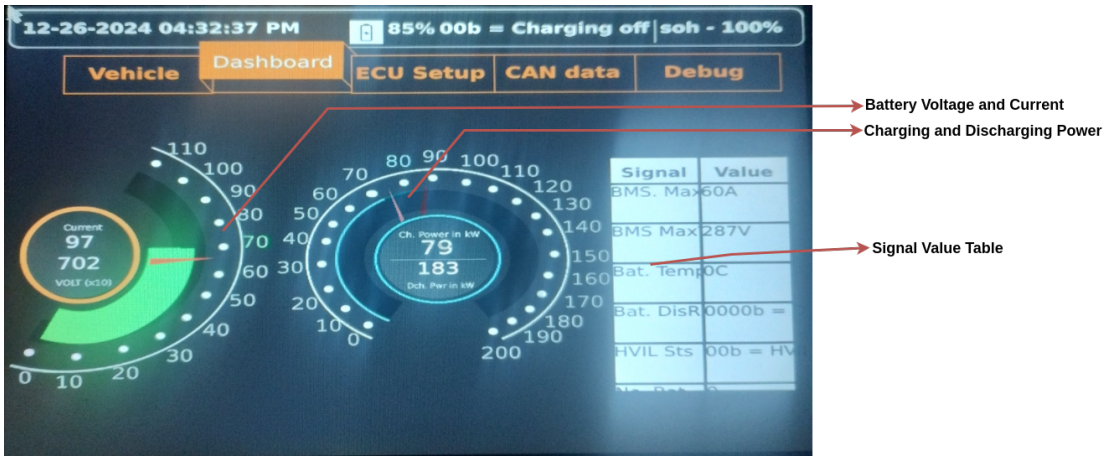

Figure 4.2: Dashboard Tab – Live Dials and Grid Display

The **Dashboard** tab is designed to provide live visualization of the most important electrical parameters (Figure 4.2). At the center are two responsive dials that display battery voltage and battery current in real time. Additional key metrics, such as power (in kilowatts), are also shown as decoded signals from the CAN network. These values are obtained directly from VIG and battery PGNs and are updated dynamically based on incoming CAN frames.

To complement the dials, the right-hand side features a live-updating signal grid that shows values such as battery temperature, HVIL (High Voltage Interlock Loop) status, contactor state, and various internal diagnostic counters. These values help operators and developers monitor operational health at a glance while also enabling real-time verification of safety-related data.

All data displayed in this tab follows the same architectural principle based on Qt's signal-slot mechanism. The backend maintains a centralized `SignalValueMap`, where each signal is stored using a standardized key format: `node.message.signal`. This structure makes it straightforward to display or update any signal in the UI with minimal additional logic. By referencing a signal using its key, the dashboard remains fully modular and can easily accommodate new signals or PGNs without requiring

structural UI changes.

4.2.4 CAN Visualizer Screen

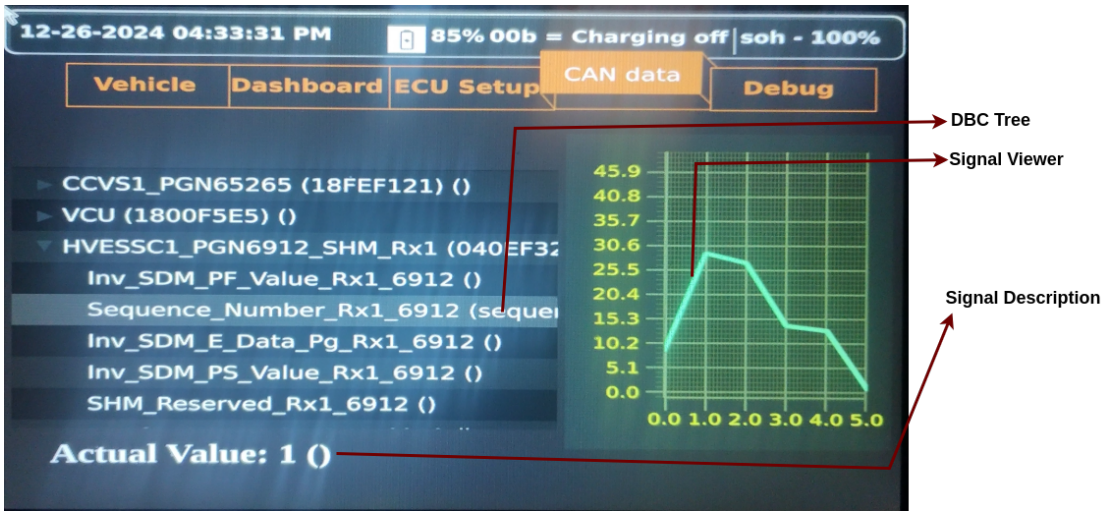

Figure 4.3: CAN Data Tab – Signal Viewer and Real-Time Plot

The **CAN Data** tab is a developer-focused diagnostic tool that aids in real-time signal analysis and trace inspection (Figure 4.3). The tab automatically parses the loaded DBC file and organizes messages and signals into a hierarchical tree view. Users can browse any PGN, drill down to its individual SPNs, and select a signal to generate a real-time plot.

The live plot shows how the selected signal evolves over the past few seconds, updating dynamically based on incoming CAN frames. The system allows configurable refresh rates and display intervals, making it suitable for debugging both slow-changing and high-frequency data. This tool has proven particularly effective in diagnosing anomalies during integration and in validating the behavior of FSM-controlled transitions and conditional signal updates.

4.2.5 Debug and System Tab

The **Debug** tab provides additional system utilities and low-level monitoring tools that assist during testing and deployment. Although not shown in a figure, this tab is critical during development and fault analysis. It displays logs generated by the embedded `LogManager`, capturing timestamped entries for events like signal mismatches, ECU communication failures, or transitions between FSM states. Developers can view the full log history and clear or export logs as needed.

In addition to logging, the Debug tab features runtime system controls such as:

- Restarting or shutting down the Raspberry Pi VCU
- Connecting to or resetting Wi-Fi configuration (useful for remote debugging or headless setups)
- Viewing system uptime, CPU load, and memory usage

These tools are essential for diagnosing issues in field deployments, especially when physical access to the system is limited. The debug tab complements the other views by offering a complete picture of backend activity, including services outside the CAN network.

4.3 ECU Setup Interface

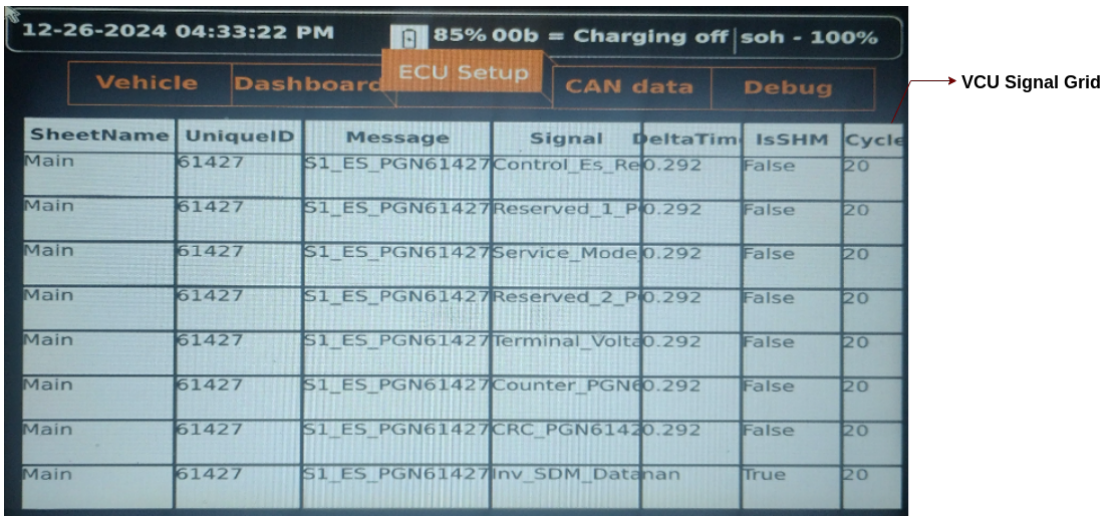

Figure 4.4: ECU Setup Tab – Signal Table View

The **ECU Setup** tab allows users to configure CAN messages and signals at runtime (Figure 4.4). This interface provides a spreadsheet-style table displaying PGNs, signal names, cycle times, and flags that determine whether a message is a Safety Header Message (SHM). This is particularly relevant for protocols like J1939-76, where message order and integrity must be strictly maintained. The interface simplifies adding or modifying messages without code recompilation, streamlining system integration and HIL testing workflows.

In addition to structural configuration, the interface also enables users to set or override signal values manually. This feature is particularly useful during debugging or simulation, where signal injection is required to validate FSM behavior, fault conditions, or ECU responses without physical input.

Furthermore, the dashboard supports a built-in **Trace Viewer** that allows users to upload a CAN log file and analyze PGNs and signal values over time. The tool decodes raw CAN messages using the same DBC file and displays the results in an interactive table and timeline-based plot. This functionality is also extended to the web via the

documentation site, allowing developers and testers to perform offline analysis or share traces remotely without requiring access to the embedded dashboard.

Additional details on the Trace Viewer is available in the *Supplementary Resources* appendix (see Appendix B).

4.4 Qt Development Workflow

The dashboard was developed using Qt Creator and Qt Designer to accelerate embedded GUI development and streamline deployment. Qt Creator served as the primary integrated development environment, offering features like real-time code debugging, Wi-Fi-based deployment to the Raspberry Pi, and support for multithreaded C++ and QML integration. Using this workflow, developers could iterate rapidly without full image rebuilds, significantly improving development efficiency.

Qt Designer, on the other hand, facilitated the visual design of QML components through a drag-and-drop interface. This reduced the learning curve and allowed quick prototyping of the UI layout. The final application integrates QML-based UI screens with C++ threads responsible for CAN communication. Signal-slot mechanisms are used to update the interface based on decoded CAN signals in real-time.

4.5 Chapter Summary

This chapter introduced the dashboard used to interact with the electrified vehicle's communication system. Developed using Qt for Embedded Linux, the dashboard integrates message decoding, signal plotting, system controls, and diagnostics into a compact and intuitive interface. With support for runtime signal configuration and real-time updates, it proved to be an essential tool during development, integration, and testing.

In the next chapter, we transition to the Hardware-in-the-Loop (HIL) testing process. This testing framework uses the dashboard as a core interface to validate CAN messaging, finite state machine behavior, and system robustness under controlled simulated environments.

Chapter 5

Hardware-in-the-Loop (HIL) Testing

5.1 Chapter Overview

This chapter discusses the HIL setup used to validate the CAN communication framework's real-time behavior in a simulated environment. The objective is to test software modules such as the finite state machine (FSM), CAN message scheduling, signal decoding, and dashboard visualization under emulated signal conditions. Fault injection and test coverage at the module level are included to ensure robustness before deployment on the physical vehicle.

5.2 HIL Setup and Configuration

Hardware-in-the-Loop (HIL) testing bridges the gap between simulation and real-world vehicle deployment. It enables closed-loop validation of embedded software components using simulated signals and actuators, without needing the physical vehicle during early development phases. Figure 5.1 shows the complete HIL testbench used for this research.

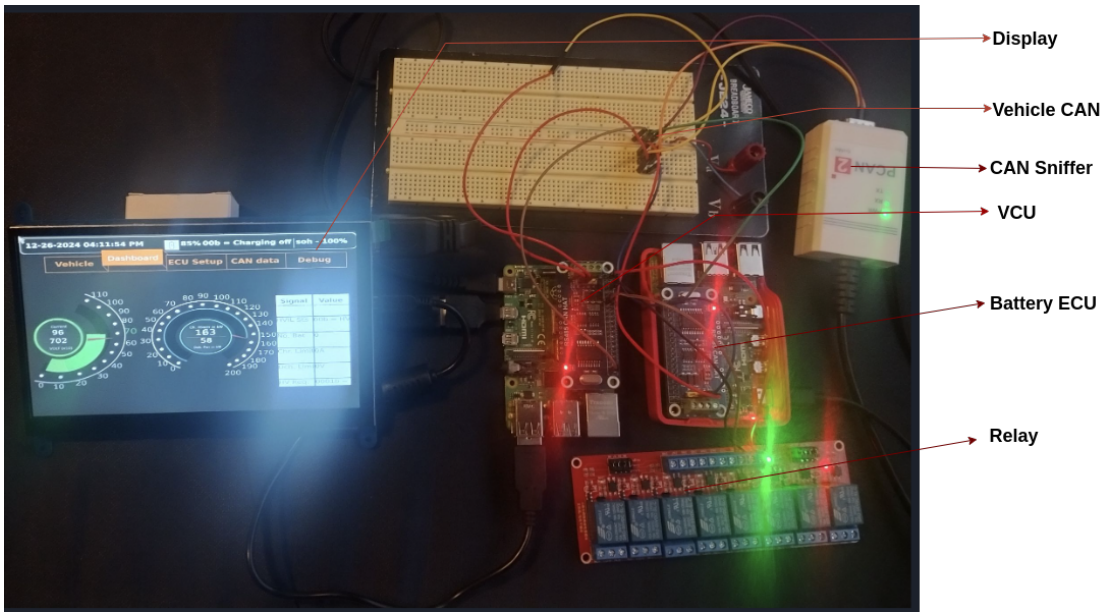

Figure 5.1: Hardware-in-the-Loop Simulation Testbench

The setup includes the following key components:

1. **Qt Dashboard with Touchscreen Display:** Acts as the Human-Machine Interface (HMI) for real-time monitoring and interaction. It visualizes decoded CAN signals, FSM states, and system status using a custom GUI.

2. **Breadboard with Signal Emulation:** Used to emulate GPIO-based sensor inputs or faults to trigger state transitions in the FSM. This allows testing different conditions such as contactor stuck, overtemperature, or charging enabled.

3. **PCAN-USB Adapter:** Provides a physical CAN interface for monitoring traffic and injecting test frames. It is connected to a host PC for debugging via tools like PCAN-View or custom Python scripts.

4. **Battery ECU (Raspberry Pi):** A second Raspberry Pi simulates the Battery Management System (BMS), responding to CAN signals like state of charge (SOC), temperature, and voltage thresholds. This allows end-to-end testing of inter-ECU communication using J1939.

5. **VCU (Raspberry Pi):** The primary controller under test, responsible for handling CAN communication, managing FSM transitions, and broadcasting relevant PGNs over the CAN bus. It runs the core logic of the CAN framework and reacts to inputs in real time (refer to Appendix A.3 for details on PGNs and SPNs)

6. **Relay Driver Board:** Connected to the VCU, this board is used to actuate physical loads such as indicator lights, contactors, or cooling systems. It enables hardware output verification corresponding to FSM state or decoded signals.

5.3 Signal Emulation and FSM Validation

Each input signal was associated with specific FSM transitions, enabling test-driven development of state logic. Scenarios such as "Battery SOC Low," "Contactor Fault," or "Inverter Ready" were injected via GPIO toggling on the breadboard. The FSM responded with appropriate state transitions such as "Charging," "Run," or "Fault Lockout."

5.4 CAN Message Testing and Timing Accuracy

CAN messages were transmitted and received through a loopback setup using the PCAN adapter and Raspberry Pi CAN drivers. Messages from the Battery ECU were parsed and verified by the VCU, while outgoing status messages were logged to check for correct update frequency and encoding. The dashboard displayed real-time values for visual inspection.

5.5 Test Scenarios and Observations

To systematically verify the functionality and robustness of the system, a set of test scenarios were defined. Each test targeted a specific aspect of the FSM logic, CAN signal flow, or output behavior. Table 5.1 summarizes key test cases executed using the HIL setup.

Table 5.1: HIL Test Scenarios and Outcomes

Test Case	**Input Condition**	**Expected Output**	**Pass/Fail**
Low SOC	SOC <20%	FSM enters Charging state	Pass
Contactor fault	GPIO fault line HIGH	FSM enters Fault state	Pass
CAN timeout	No battery ECU PGNs for 3s	FSM triggers timeout warning	Pass
Relay control	FSM in RUN state	Output relay toggled ON	Pass

5.6 Dashboard Interaction and Debug Features

The Qt-based dashboard supported multiple tabs including signal monitoring, raw CAN data logs, and debug information for state transitions. This allowed developers to validate the internal logic, tune timing parameters, and visualize how faults or conditions affected the control flow.

5.7 Chapter Summary

This chapter established a comprehensive hardware-in-the-loop simulation environment to validate the proposed CAN communication framework, FSM logic, and dashboard performance. With the Battery ECU and VCU communicating over a shared CAN bus and signals emulated via GPIO, the system was thoroughly tested in a safe and reproducible manner. In the next chapter, we transition from simulated testing to vehicle integration and real-world deployment, applying the framework to the electrified compact track loader prototype to evaluate its functionality in an actual operating environment.

Chapter 6

Vehicle Integration and Real-World Testing

6.1 Chapter Overview

This chapter presents the integration and real-world testing of the electrified compact track loader, emphasizing how individual subsystems interact and function cohesively under real operating conditions. It begins with a high-level overview of the vehicle's physical and electrical architecture, setting the stage for detailed exploration of battery communication and initialization processes. These include the enablement of J1939 communication, configuration of the high-voltage topology, and execution of start-up and shutdown sequences.

Particular attention is given to CAN-based messaging strategies, such as battery keep-alive messages and the generation of Safety Header Messages, which ensure safe and reliable power distribution across components. The behavior of battery connection signals is also analyzed to understand temporal dependencies and transitions during system activation.

Subsequent sections detail the communication sequences of other critical subsystems including the Power Distribution Unit (PDU), Editron inverter, and HVLP inverters, outlining how these components are orchestrated through message-based control. These integration efforts validate the framework's robustness and readiness for deployment, bridging the gap between software-in-the-loop testing and full-scale vehicle operation in

the field.

6.2 Vehicle System Overview

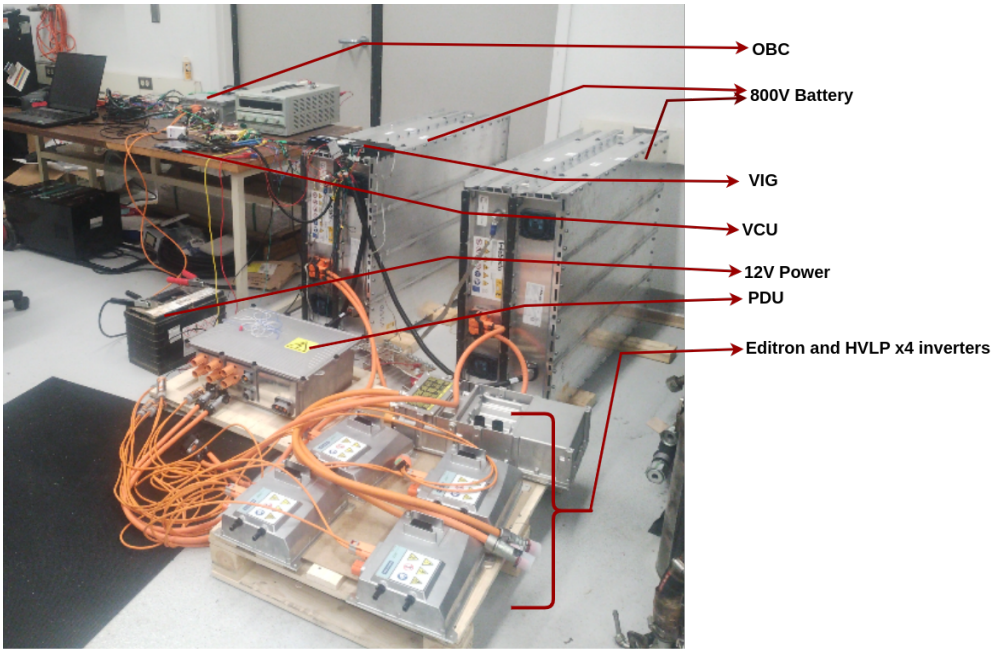

Figure 6.1: Integrated component setup for real-world testing

The transition from software-in-the-loop (SIL) and hardware-in-the-loop (HIL) testing to a complete physical integration marks a critical milestone in the validation of the proposed framework. The image shown in Figure 6.1 captures the real vehicle hardware setup used for validation. This setup included the high-voltage battery pack, inverter, Vehicle Control Unit (VCU), power distribution components, and various auxiliary systems.

At this stage, the control and communication frameworks developed in earlier phases were deployed on actual hardware. The complete CAN-based system, driven by J1939 protocol, was brought online to validate real-time behavior. Rather than re-explaining each component, the emphasis in this chapter is on how these modules interacted in practice—through the lens of PGN exchanges, timing, signal integrity, and subsystem

coordination.

The battery was central to this process—not only as the power source but as a smart unit requiring precise sequencing and feedback management. Communication between the VCU and the Battery Management System (BMS) was implemented over the J1939 protocol, with critical PGNs such as State of Charge (SoC), State of Health (SoH), Pack Voltage, Current, and Contactors Status being periodically broadcast and consumed by the system.

Initial power-up involved orchestrating a pre-charge sequence, verifying signal acknowledgment from the BMS, and engaging the main contactors. The VCU sent enable commands and waited for voltage stabilization feedback from the battery before moving to the operational state. These transitions were monitored through signal traces to ensure proper timing and response, as will be discussed in the following sections.

This integration effort provided essential insights into real-time behavior of the electrified powertrain, highlighting both expected performance and unanticipated edge cases. These findings, along with CAN signal logs, PGN-specific behavior, and test results, are detailed in the subsequent sections.

6.3 Battery Communication and Initialization

Bringing the battery system online required more than just connecting cables and applying power—it involved a carefully sequenced set of CAN communications, safety handshakes, and control logic initialization to ensure safe and deterministic behavior. The Webasto VIG (Vehicle Interface Gateway) served as the bridge between the high-voltage battery pack and the rest of the vehicle, and its configuration played a critical role in enabling J1939-based communication.

The overall CAN initialization flow for the Webasto VIG is illustrated in Figure 6.2, highlighting the step-by-step process necessary to bring the battery system online.

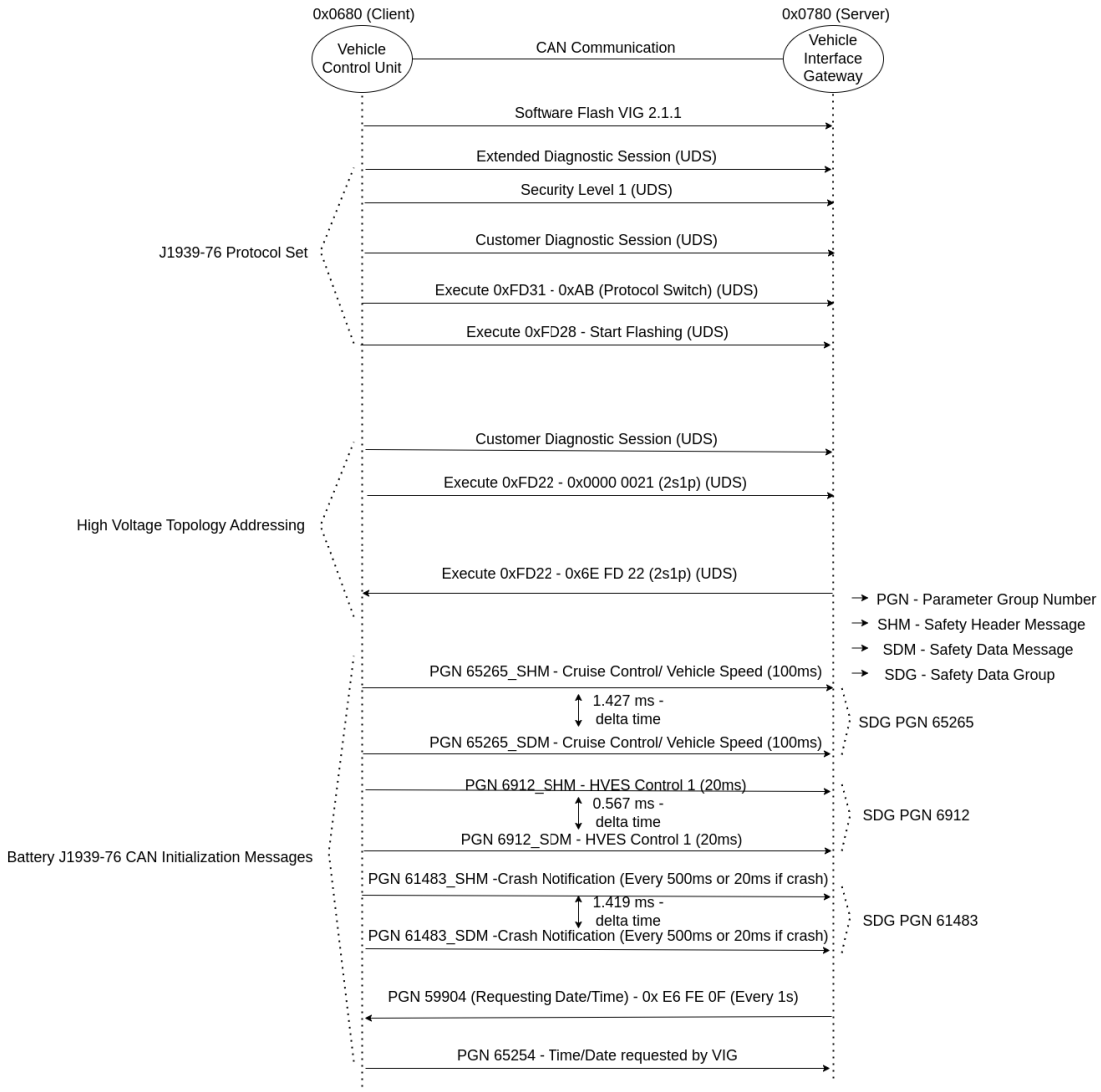

Figure 6.2: Webasto CAN Initialization Flow

The initialization sequence was divided into three stages: enabling J1939 communication using UDS over CAN, setting the appropriate high-voltage topology, and maintaining system-level communication via periodic PGNs. Each step was essential for establishing reliable communication between the VCU and the battery management system (BMS), ensuring safe and consistent operation in real-world conditions.

6.3.1 Enabling J1939 Communication

By default, the VIG operates using the CAN 2.0 protocol. To integrate with the J1939-based vehicle network, the VIG had to be reconfigured. This was accomplished through a Unified Diagnostic Services (UDS) session initiated by the VCU. In this client-server model, the VCU acts as the client (ID: 0x680), and the VIG responds as the server (ID: 0x780).

The UDS protocol configuration process is shown in Figure 6.3. It involves several diagnostic session transitions and secure authentication steps before enabling J1939.

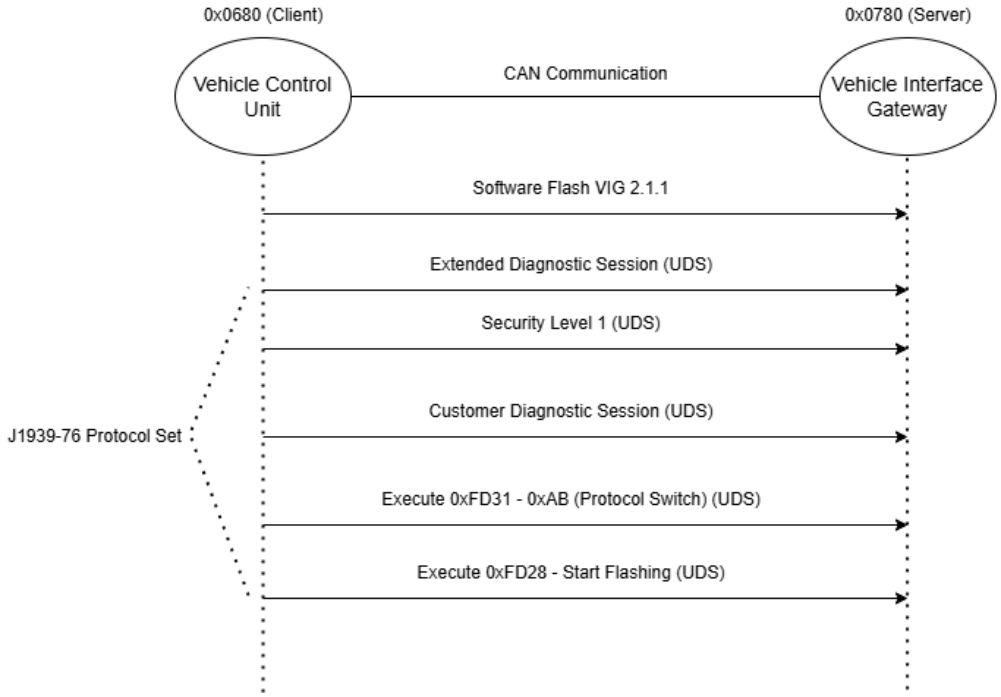

Figure 6.3: UDS Protocol Configuration

The process began with flashing the VIG using the V-Flash tool to ensure it was running the correct firmware. After flashing, the VCU initiated an Extended Diagnostic Session using a UDS request. It then transitioned into the Customer Diagnostic Session through a vendor-specific seed-key authentication mechanism: the VCU first requested a seed from the VIG, then used a vendor-provided dynamic library to generate the

matching key, which was sent back to unlock protected functions.

Once in the Customer Diagnostic Session, the VCU issued a configuration command to switch the protocol to J1939-76. This configuration was written into the VIG's memory, enabling it to persist through reboots and begin participating in the J1939 communication network upon startup.

6.3.2 High Voltage Topology Configuration

After successfully enabling J1939 communication, the next critical step involved configuring the high-voltage electrical topology of the battery. As shown in Figure 6.4, a 2s1p (two cells in series, one in parallel) configuration was selected for the current system setup. This topology was transmitted to the VIG using the diagnostic data value `0021`. Upon successful reception, the VIG issued a positive acknowledgment message (`0x6E FD 22`), confirming that the high-voltage configuration had been correctly applied. This step was essential to ensure that the Battery Management System (BMS) could internally interpret series and parallel arrangements accurately.

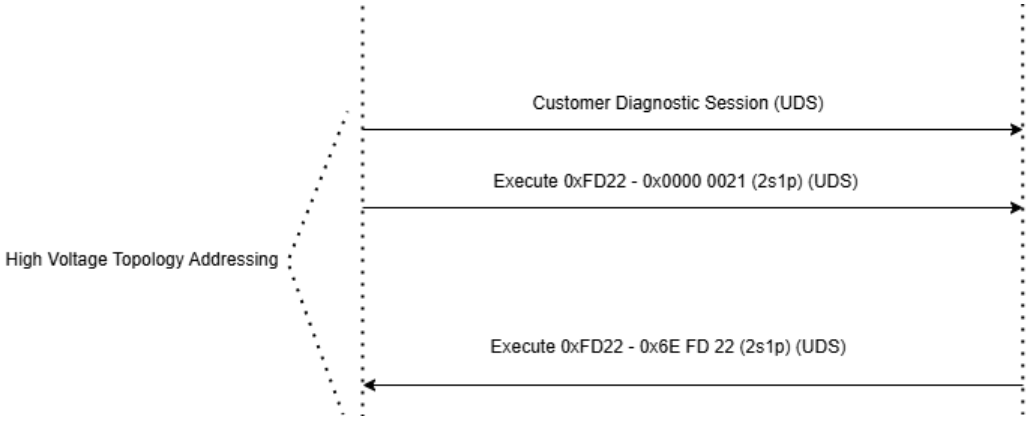

Figure 6.4: High Voltage Topology Setting

6.3.3 CAN Messaging and Battery Keep-Alive

As visualized in Figure 6.5, maintaining the operational state of the battery system requires continuous transmission of specific J1939 Parameter Group Numbers (PGNs).

These messages act as a heartbeat for the system, ensuring that the VIG and other connected ECUs remain synchronized and fault-free.

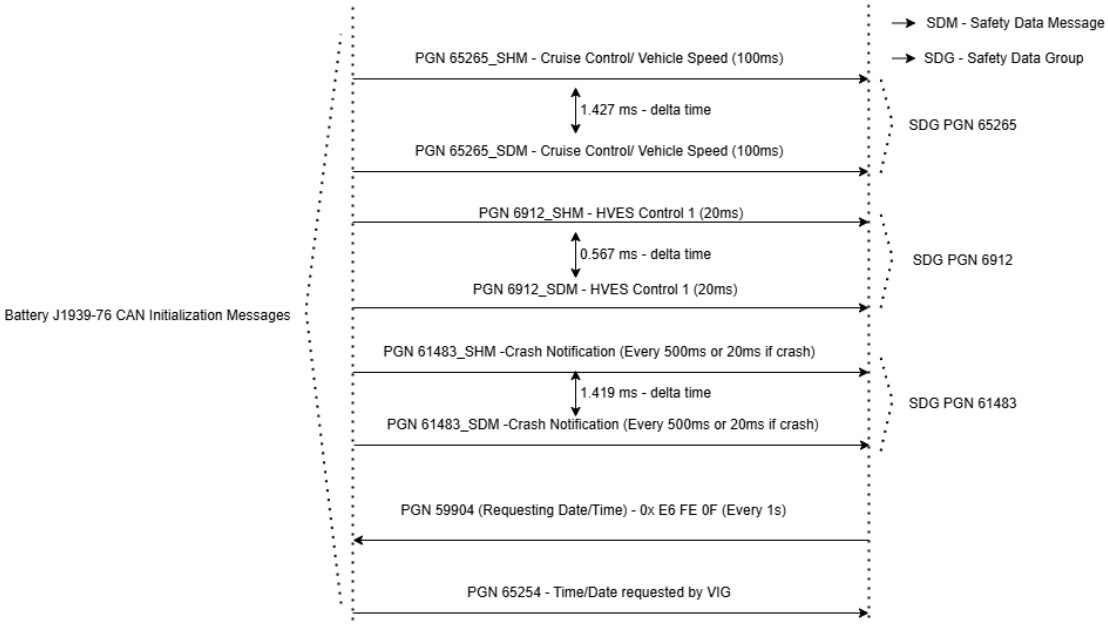

Figure 6.5: CAN Message Initialization Sequence

PGN 65265 – Cruise Control / Vehicle Speed

One such keep-alive message is **PGN 65265**, typically associated with cruise control and vehicle speed data. In this system, it is repurposed to signal operational readiness and keep the battery in an active communication state. The message is transmitted every 100 milliseconds.

The signal structure for PGN 65265 is detailed in Table 6.1. While most of the individual signal values are placeholders or unused in this context, their presence is still mandatory according to the battery's firmware expectations.

Table 6.1: PGN 65265 Cruise Control/Vehicle Speed

Signal	**Value**	**Description**
Two_Spd_Axle_Switch Parking_Brake_Swtch Cruise_Control_Pause_Swtch Park_Brake_Rel_Inhibit_Req Cruise_Control_Active Cruise_Control_Enable_Swtch Brake_Switch Clutch_Switch Cruise_Control_SetSwtch Cruise_Control_Coast_Decl_Switch Cruise_Control_Resume_Swtch Cruise_Control_Accelerate_Swtch Engine_Idle_Increment_Switch Engine_Idle_Decrement_Switch Engine_Diag_Test_Mode_Swtch Engine_Shutdwn_Ovride_Swtch	3	Not Available or Take No Action
Wheel_Based_Veh_Spd PTO_Governor_State	0	Off/Not Set
Cruise_Control_States	7	Not Available
Cruise_Control_Set_Spd	255	Desired Value

The continuous broadcasting of this PGN ensures the battery interprets the system as active, avoiding undesired timeouts or shutdown conditions. While the data fields themselves may not affect drive control directly in this application, their presence is still vital to fulfill protocol requirements for keep-alive logic.

PGN 6912 – HVES Control 1

The HVES Control 1 message, identified by **PGN 6912**, plays a central role in the high-voltage battery's operational workflow. As illustrated in Table 6.2, this message handles several critical battery control operations including charger enablement, isolation test initiation, contactor commands, and operational consent for high-voltage components.

Table 6.2: PGN 6912 HVES (High Voltage Energy System) Control 1

Signal	**Value**	**Description**
Two_Spd_Axle_Switch HS_HiUBusCnctCmd_Rx1 HS_PwrDwnCmd_Rx1 HS_HiUBusAcvIslnTestCmd_Rx1 HS_CellBalnCmd_Rx1 HS_EnaIntChrgrCmd_Rx1 HS_HiUBusHiSideRestrCnctReq_Rx1 HS_HiUBusLoSideRestrCnctReq_Rx1	3	Not Available or Take No action
HS_HiUBusPasIslnTestCmd_Rx1 HS_OperConsent_Rx1	1	Isolation Test On; HVES Consent given
HS_Ctl1Ctr_Rx1s	15	Fixed Value. Only used in Hybrid mode
HS_Ctl1CRC_Rx1	255	Fixed Value. Only used in Hybrid mode

This message is broadcast at a fixed interval of 100 ms, ensuring real-time responsiveness for battery control operations. Its reliable presence ensures consistent battery readiness and supports critical safety operations such as insulation monitoring and charger coordination.

PGN 61483 – Crash Notification

For safety-critical scenarios, **PGN 61483** provides real-time crash notification data. Even in non-crash situations, this message must be broadcast every 500 ms to maintain system-wide synchronization. It allows connected ECUs to continuously evaluate crash state and execute fallback or isolation logic accordingly.

Table 6.3 outlines the structure of this message. While the crash counter and checksum values are generally unused in J1939-76 mode, the `Crash_Typ` field must be explicitly set to `0` to indicate a normal operating state.

Table 6.3: PGN 61483 Crash Notification

Signal	**Value**	**Description**
Crash_Ctr Crash_Cks	Any number	Not used in J1939-76 mode
Crash_Typ	0	No Crash

In the case of an actual crash, or under-voltage events like a Cl30C supply drop, the battery management system may trigger a permanent or temporary lockout of contactors. The crash notification PGN serves to alert all ECUs of this state, ensuring coordinated shutdown procedures and battery safety enforcement.

PGN 65254 – Date and Time

Another essential part of the system is time synchronization. PGN 65254 is requested once per second by the VIG using PGN 59904 (Request Message). This message helps synchronize the date and time across all ECUs in the vehicle. Accurate timestamping is critical for diagnostics, event tracking, and data logging—especially during debugging of intermittent faults or correlating trace logs with system states.

6.3.4 Battery Start Sequence

Bringing the battery online involves a structured and safety-compliant sequence of CAN messages exchanged between the VCU and the Vehicle Interface Gateway (VIG). This

process ensures safe activation of high-voltage systems and provides a robust handshake before the contactors close.

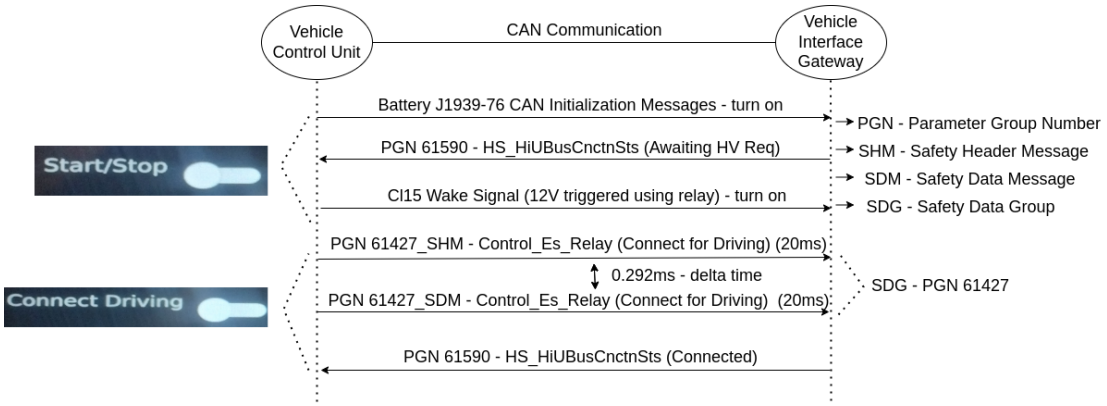

Figure 6.6: Battery Start Sequence

The sequence begins when the operator presses the Start/Stop switch on the vehicle dashboard. This triggers the periodic transmission of initialization messages over the CAN network. The VIG then responds with a message on PGN 65190, reporting the signal `HW_HiUBusCnctnSts` as "Awaiting HV Req." This indicates that the battery is awake but not yet permitted to connect its high-voltage bus.

Internally, the system now enables the Cl15 wake signal via a relay, activating internal circuits. Once this state is achieved, the user proceeds by activating the "Connect Driving" switch on the dashboard. This sends PGN 61427 from the VCU, setting the `Control_Es_Relay` signal to a value of `3`—which corresponds to "Connect for Driving."

The VIG then evaluates the request and, if all safety and logical conditions are satisfied, responds on PGN 61590 with a status indicating "Connected." At this point, the high-voltage contactors close, and the system is ready for operation, as shown in Figure 6.6. The signals used for this process are described in Table 6.4.

Table 6.4: PGN 61427 HVES Ignition

Signal	Value	Description
ShutDown_Clearance Service_Mode_Request	3	Not Available
Counter_PGN61427 CRC_PGN61427	15	Fixed Value
Control_Es_Relay	3	Connect for Driving
Reserved_1_PGN61427 Reserved_2_PGN61427 Terminal_Voltage_Actual	1 4294967295 2047.5	Reserved Reserved Actual value of terminal voltage before main contactor

6.3.5 Battery Shutdown Sequence

Battery shutdown is handled in a safe and controlled manner to ensure all high-voltage paths are disconnected and the system enters a low-power state. The process reverses the start-up sequence while monitoring feedback from the VIG to confirm successful disconnection.

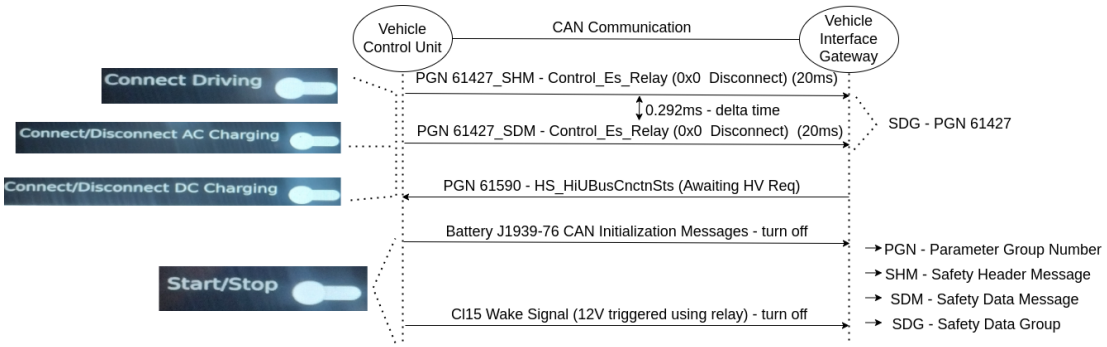

Figure 6.7: Battery Shutdown Sequence

The shutdown is initiated when the user deactivates the “Connect for Charging” or “Connect for Driving” switch on the dashboard. This sends an updated PGN 61427

message with a Disconnect value on the `Control_Es_Relay` signal. The VCU then waits for the VIG to respond on PGN 61590, with a new status indicating "Awaiting HV Req"—a clear confirmation that the high-voltage bus has been disconnected.

Once the disconnection is confirmed, the Start/Stop switch can be turned off, stopping the CAN initialization message loop. The Cl15 wake signal is also deactivated automatically, completing the battery shutdown sequence (Figure 6.7).

6.3.6 Battery - OBC AC Charging Sequence

The battery charging process in AC mode relies on orchestrated messaging between the VCU, Vehicle Interface Gateway (VIG), and the On-Board Charger (OBC). The sequence outlined in Figure 6.8 illustrates the key steps required to activate AC charging, monitor limits, and engage the charger.

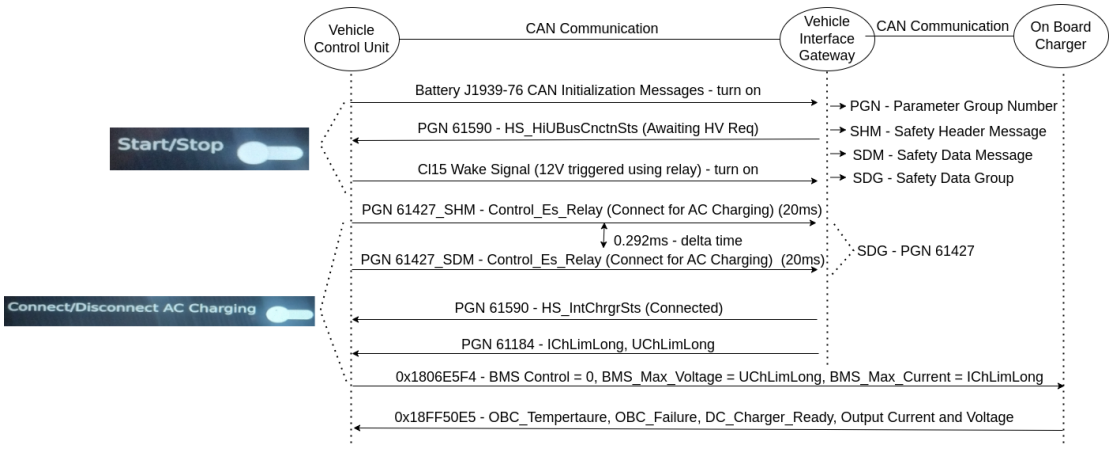

Figure 6.8: Battery – OBC AC Charging Sequence

The sequence starts when the user presses the Start/Stop switch, initiating standard CAN initialization messages. In response, the VIG broadcasts PGN 65190 with the signal `HW_HiUBusCnctnSts` indicating *Awaiting HV Req.* At the same time, the Cl15 wake signal is asserted to power on internal circuitry.

To begin AC charging, the user activates the "Connect AC Charging" switch on the dashboard. This triggers the VCU to send a message on PGN 61427, setting the

`Control_Es_Relay` signal to *Connect for AC Charging*. The VIG replies with PGN 61590, where the signal `HS_IntChrgrSts` confirms readiness for charging.

Following this handshake, the battery voltage and current limits are extracted from PGN 61184 and forwarded to the OBC by the VCU. The `BMS_Control` signal is set to `0`, which enables AC charging mode. The OBC responds with actual measured values such as output voltage, current, and internal state, allowing the dashboard to visualize real-time AC charging progress and system health.

6.3.7 Battery - OBC DC Charging Sequence

DC charging follows a similar communication path but utilizes a higher-current command structure for faster charging operations. The control sequence is shown in Figure 6.9.

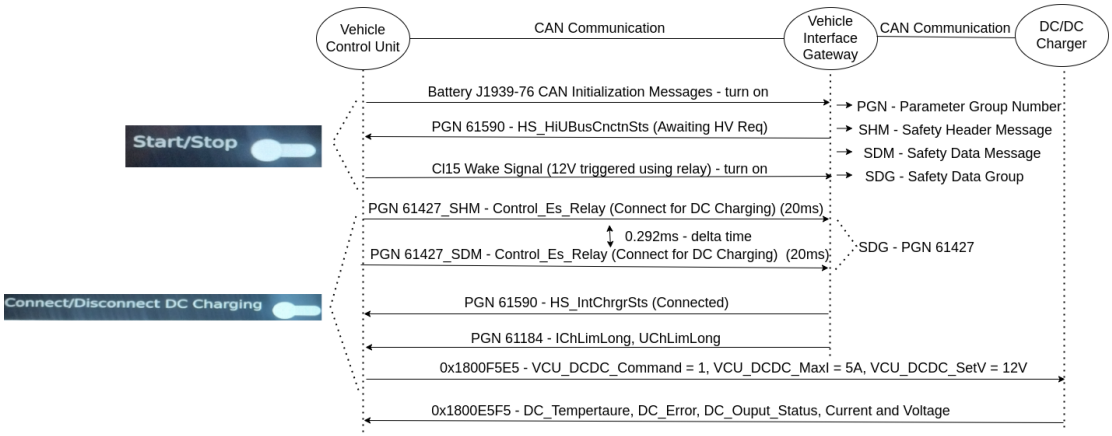

Figure 6.9: Battery – OBC DC Charging Sequence

The process begins with activation of the battery system via the Start/Stop switch. Initialization messages are broadcast, and the VIG indicates its standby state by transmitting PGN 65190 with the `HW_HiUBusCnctnSts` signal set to *Awaiting HV Req.* The Cl15 line is also powered on.

The user then selects “Connect DC Charging” from the dashboard, prompting the VCU to send PGN 61427 with the `Control_Es_Relay` set to *Connect for DC Charging*. The VIG replies on PGN 61590, confirming readiness with the `HS_IntChrgrSts` signal.

Once readiness is confirmed, the VCU sends DC charging limits (voltage and current) to the OBC using the `VCU_DCDC_Command` signal. These values are derived from battery capability data received earlier. Upon accepting the command, the OBC activates the charger and transmits status frames containing real-time voltage, current, and fault states, ensuring reliable DC charging and visibility on the dashboard.

6.3.8 Safety Header Message Generation

Certain messages within the battery communication framework—particularly under the J1939-76 protocol (Type 1)—require the transmission of a **Safety Header Message (SHM)** followed by a **Safety Data Message (SDM)**. This layered message structure enhances data integrity by appending a checksum and metadata to each message, providing fault detection capabilities critical for high-voltage battery systems.

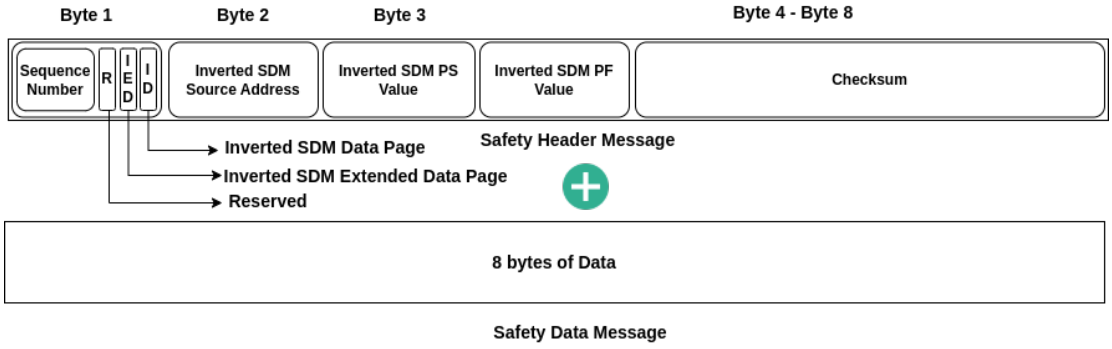

Figure 6.10: Representation of SHM and SDM structure

For each PGN corresponding to a safety-critical message, two CAN frames are transmitted:

- `S1_ES_PGN61427` – the SDM, containing the actual payload
- `S1_ES_PGN61427_SHM` – the SHM, constructed based on the SDM content

Figure 6.11 illustrates a typical message exchange between the VCU and the battery management system (BMS), with SHM and SDM being sent in succession.

S1_ES_PGN61427				
SDM				
Cycle Time	Signals	CAN ID	Value	Data
20.193819		CEFF321		EF FF FF FF FF FF FF FF
	ShutDown_Clearance		3	
	Control_Es_Relay		3	
	Reserved_1_PGN61427		1	
	Service_Mode_Request		3	
	Reserved_2_PGN61427		FFFFFFFF	
	Terminal_Voltage_Actual		FFF	
	Counter_PGN61427		F	
	CRC_PGN61427		FF	
S1_ES_PGN61427_SHM_Rx				
SHM				
Cycle Time	Signals	CAN ID	Value	Data
20.193819		C0EF321		7F DE 0C 10 A3 EF 00 81
	Inv_SDM_Data_Pg_Rx_61427		1	
	Inv_SDM_E_Data_Pg_Rx_61427		1	
	SHM_Reserved_Rx_61427		1	
	Sequence_Number_Rx_61427		F	
	Inv_SDM_SA_Rx_61427		DE	
	Inv_SDM_Data_PS_Value_Rx_61427		C	
	Inv_SDM_Data_PF_Value_Rx_61427		10	
	SDM_Data_CRC_Rx_61427		8100EFA3	

Figure 6.11: Example of SDM followed by SHM transmission

The process begins with constructing the SDM. This involves populating defined signal fields based on the PGN's message structure. Figure 6.12 outlines this preparation stage.

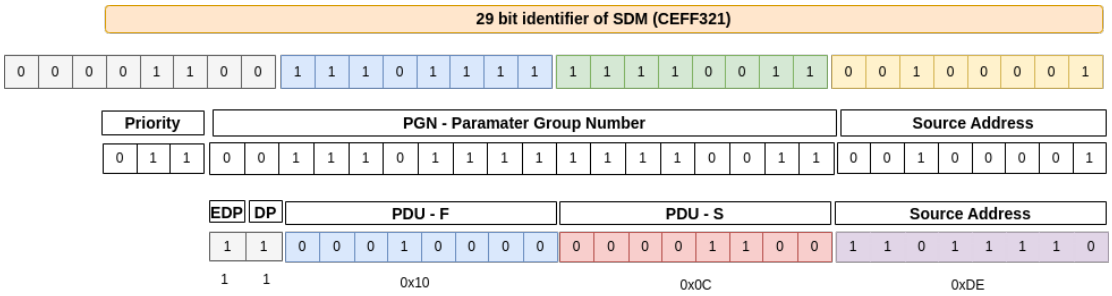

Figure 6.12: SHM Generation: Step 1 – SDM Construction

Construct Initial Signals: For a PGN such as 61427, the following internal fields are created during SDM formation:

- `Inv_SDM_Data_Pg_Rx_61427`
- `Inv_SDM_E_Data_Pg_Rx_61427`
- `SHM_Reserved_Rx_61427`
- `Inv_SDM_SA_Rx_61427`
- `Inv_SDM_PS_Value_Rx_61427`
- `Inv_SDM_PF_Value_Rx_61427`

Once the SDM is fully assembled, the next step is to generate a checksum for the SHM. This checksum authenticates the previously transmitted SDM, allowing the receiving BMS to validate message content before processing.

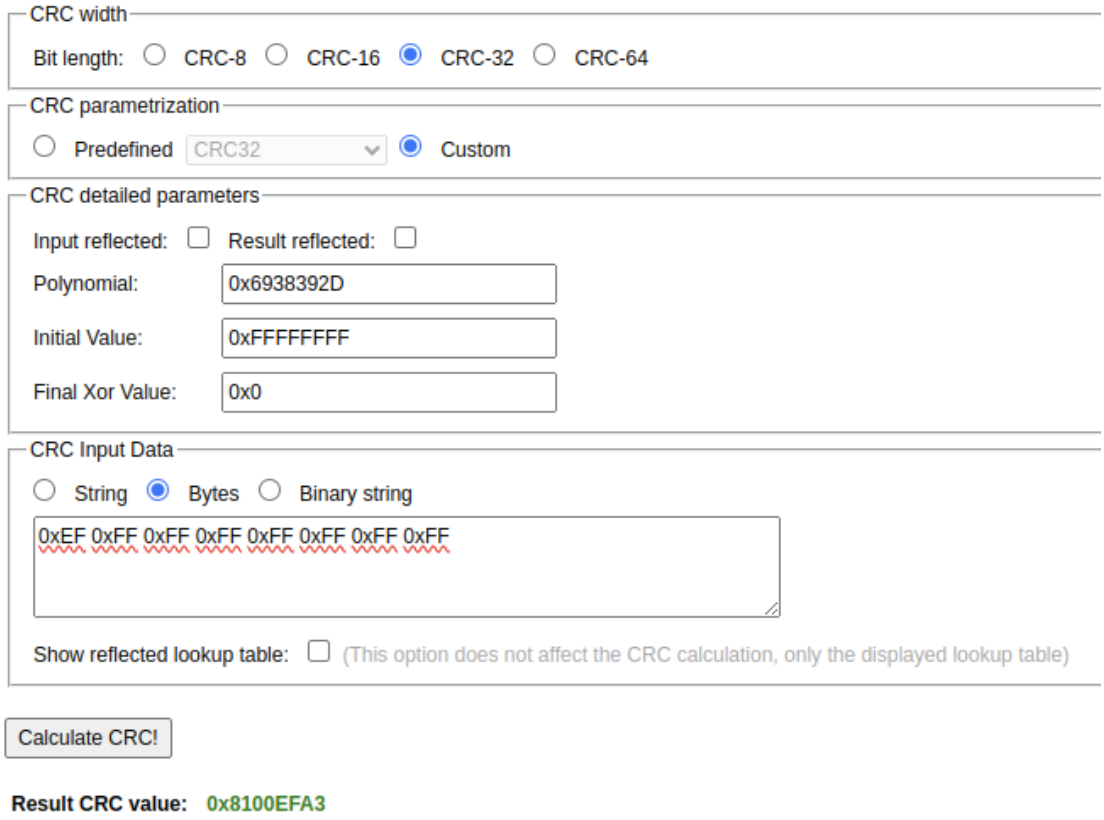

Figure 6.13: SHM Generation: Step 2 – Checksum Calculation

Generate Checksum: A 32-bit CRC checksum is calculated using the polynomial `0x6938392D`. The software-side implementation uses the `Crc32Calculator` class to compute this checksum. Two functions in the source code handle the full transmission cycle:

- `CreateSDMData()` – prepares the Safety Data Message payload
- `CreateSHMData()` – generates the Safety Header Message, including the CRC checksum

The SHM is transmitted after a protocol-defined time delta following its corresponding SDM. This sequencing requirement ensures deterministic behavior, allowing the BMS to associate each SHM with its preceding SDM and validate the message contents

in real time.

6.3.9 Battery Connection Signal Behavior

Real-time testing provided critical insight into signal transitions during the battery connection phase. The plots in Figures 6.14 and 6.15 are extracted from CAN trace logs, illustrating the behavior of key signals involved in initiating and validating high-voltage bus connection.

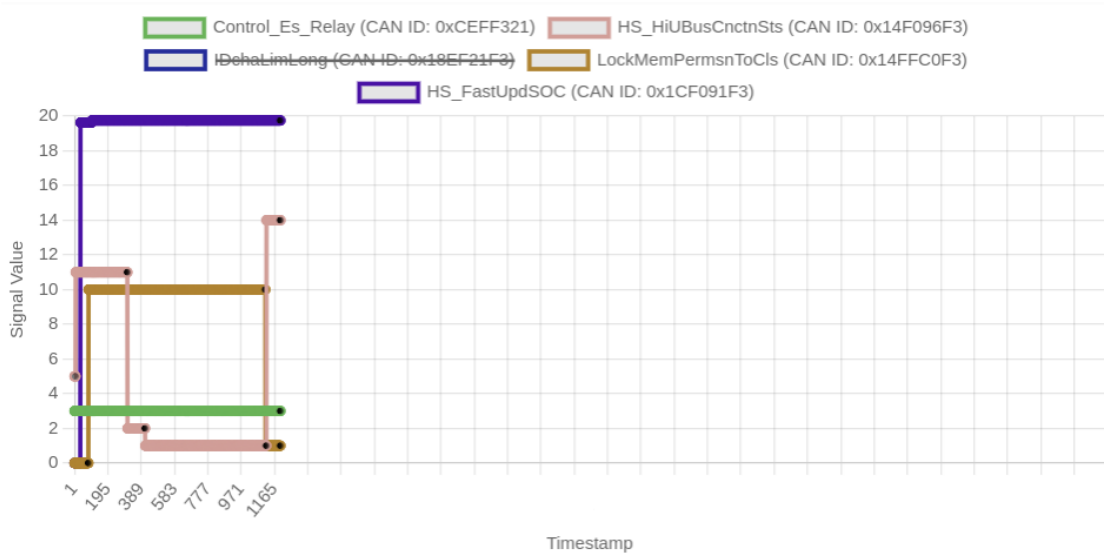

Figure 6.14: Battery Connection Signals – Initial Contact Request

As shown in Figure 6.14, when the operator presses the *Connect Driving* button on the dashboard, the signal `Control_Es_Relay` is set to value `3`, which issues a command to close the high-voltage contactors. Upon receiving this command, the Vehicle Interface Gateway (VIG) updates the signal `HS_HiUBusCnctnSts` to value `1`, indicating successful connection of the high-voltage bus.

Simultaneously, the signal `PermissionToClose` transitions to value `10`, confirming that the system permits contactor closure. The `FastUpdSOC` signal also registers a state of charge (SOC) of `19%`, indicating that the battery is active and partially charged prior to engaging the load.

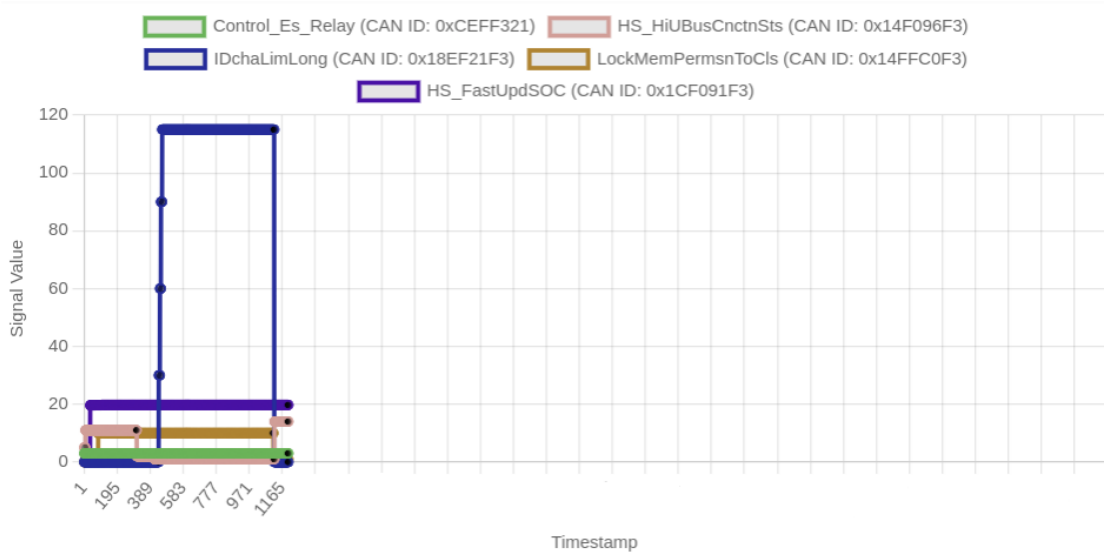

Figure 6.15: Battery Connection Signals – Post-Connection Load Behavior

As seen in Figure 6.15, once the contactors are closed, the discharge current rapidly ramps up to approximately `110 A`. Despite the SOC remaining at `19%`, the system demonstrates sufficient capacity to support high current draw immediately after connection, highlighting its ability to perform under partial charge.

These trace plots validate the full control loop implemented in the VCU—from user input and CAN message dispatch to acknowledgment and real-world load response. The synchronized transitions across `Control_Es_Relay`, `HS_HiUBusCnctnSts`, and current measurements confirm that the FSM logic and message handling framework reliably enable high-voltage engagement in dynamic scenarios.

6.4 Power Distribution Unit (PDU) Sequence

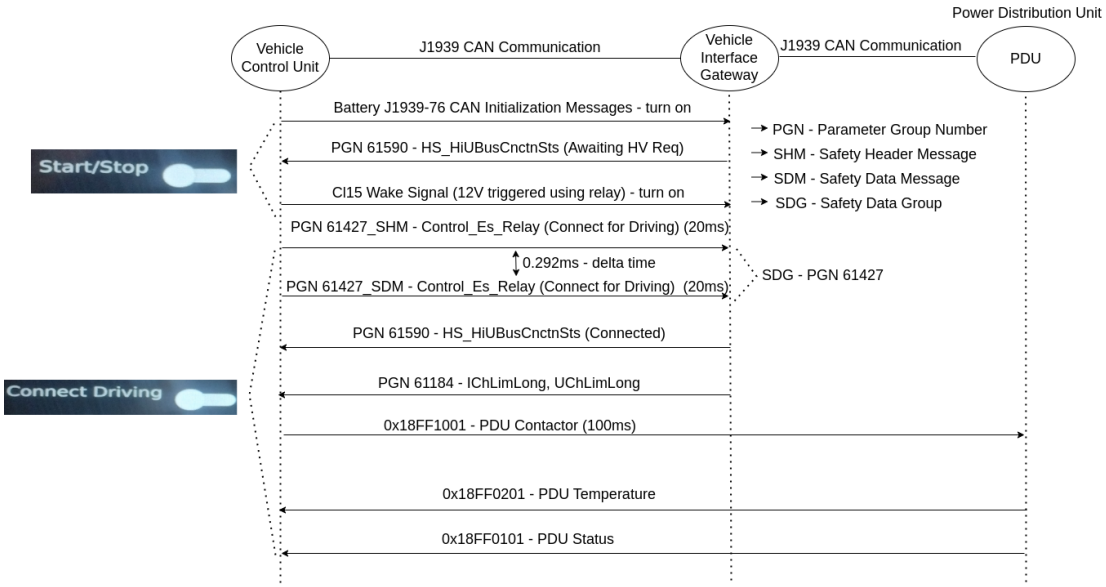

Figure 6.16: PDU Activation Sequence

The Power Distribution Unit (PDU) serves as a critical intermediary, distributing high-voltage power from the battery to downstream components such as the inverters. During system integration testing, the PDU demonstrated reliable coordination with the battery's contactor logic.

As shown in Figure 6.16, once the `Control_Es_Relay` command was transmitted by the VCU and acknowledged by the Vehicle Interface Gateway (VIG), the PDU contactors were activated in parallel. This parallel operation was reflected in the CAN trace logs, where the timing of PDU energization closely followed the confirmation of high-voltage bus connection.

This behavior validated the deterministic characteristics of the system's J1939-based communication framework, confirming that the PDU's response did not exhibit unexpected delays or sequencing faults during high-voltage transitions.

6.5 Editron Inverter Communication Sequence

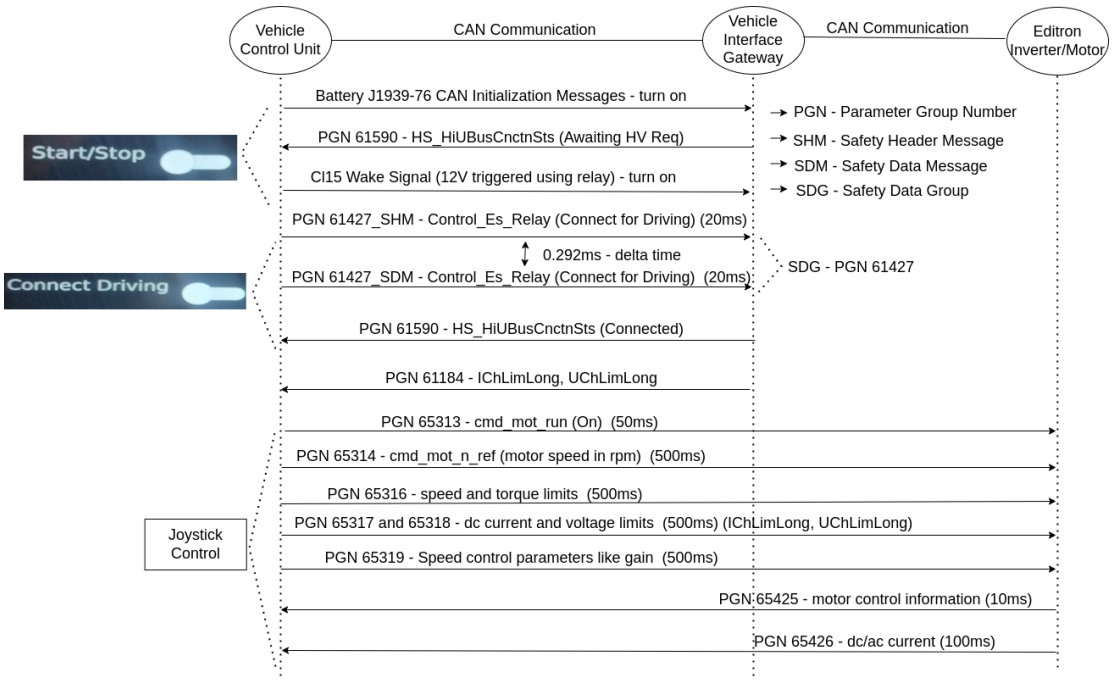

Figure 6.17: Editron Inverter CAN Communication Sequence

Following successful activation of the high-voltage bus and PDU, the Editron inverter communication sequence is initiated. As illustrated in Figure 6.17, this process conforms to the J1939-76 protocol, with communication structured around cyclic PGN-based message groups.

The Editron inverter is responsible for controlling a single dedicated Editron motor. It is the only inverter in the system that communicates over the main CAN bus using standard J1939-76 messages, making it distinct from the Parker HVLP inverters, which operate on a separate CAN line with a different protocol stack.

Upon confirmation of connection via the `HS_HiUBusCnctnSts` signal, the VCU transmits PGN 61427 using the Safety Header Message (SHM) and Safety Data Message (SDM) format. This Safety Data Group (SDG) ensures message integrity and compliance with J1939-76.

Once the SHM/SDM group is validated by the inverter, additional PGNs are sent periodically to configure operational parameters:

- **65313** – Command motor run
- **65314** – Motor speed reference
- **65316** – Speed and torque limits
- **65317, 65318** – DC current and voltage limits
- **65319** – Speed control parameters

These messages are transmitted at 500 ms intervals to define the operating envelope. Feedback from the inverter is received on high-priority PGNs such as:

- **65425** – Motor control status
- **65426** – DC/AC current values

These feedback frames, sent every 10–100 ms, enable tight control loops and real-time performance updates, ensuring the VCU can make informed decisions based on actual inverter response.

6.6 HVLP Inverter Sequence

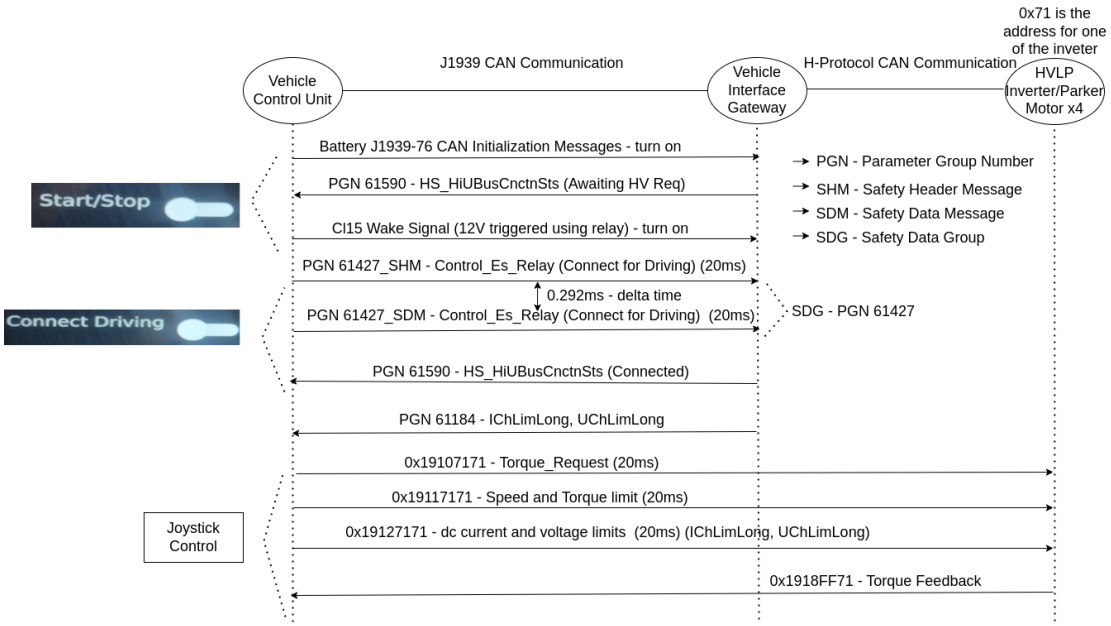

Figure 6.18: HVLP Inverter Communication Flow

The Sevcon HVLP inverter subsystem consists of four separate inverters, each responsible for driving an individual Parker motor. These inverters communicate using a custom protocol built on top of CAN—commonly referred to as H-Protocol—and are addressed uniquely via node identifiers: `71`, `72`, `73`, and `74`. In parallel, the system also includes a single Editron inverter used to drive the fifth electric motor, following the standard J1939-76 protocol.

The initialization process for each HVLP inverter begins with the Vehicle Control Unit (VCU) transmitting SHM and SDM message pairs using PGN 61427, requesting high-voltage connection permission. Once each inverter responds with an acknowledgment (`HS_HiUBusCnctnSts` = "Connected"), the VCU proceeds to read battery charge/discharge limits from PGN 61184 and begins streaming motor control messages on each CAN channel corresponding to the four HVLP inverter addresses.

The following PGNs are used for real-time motor control and are transmitted at a

high-frequency cycle time of 20 ms:

- `0x19107171` – Torque Request
- `0x19117171` – Speed and Torque Limits
- `0x19127171` – DC Current and Voltage Limits

These commands are dynamically generated by the VCU based on joystick input or FSM-determined operating mode. Each PGN is customized per inverter by embedding the correct source and destination address, allowing individual control of each motor.

In return, every HVLP inverter transmits its operational feedback on PGN `0x1918-FF71`, which includes torque output, speed reference, current draw, and error status flags. These responses are parsed in the VCU's receive loop and stored in the shared `dataVector` memory for logging, dashboard visualization, and safety checks.

As shown in Figure 6.18, this tightly looped, low-latency message exchange validates the VCU's ability to manage multiple CAN nodes with distinct IDs in parallel. Furthermore, the system architecture allows the same VCU to handle both H-Protocol (HVLP) and J1939-76 (Editron) communication stacks concurrently, demonstrating the modularity and protocol abstraction achieved by the designed software framework.

6.7 Chapter Summary

This chapter presented the real-world integration of the electrified compact track loader, transitioning from simulation and HIL environments into full hardware deployment. Key components including the battery system, Vehicle Interface Gateway (VIG), Power Distribution Unit (PDU), and both HVLP and Editron inverters were configured and tested within a live setup using the J1939-76 protocol framework.

Through detailed walkthroughs of start-up, shutdown, and charging sequences, it was shown how the Vehicle Control Unit (VCU) orchestrated system-level behavior by transmitting carefully timed and structured PGNs. The implementation of Safety Header Messages (SHM) and Safety Data Messages (SDM) was validated both in theory and in real signal traces. Specific attention was given to how signal feedback (e.g., contactor state, SOC, current draw) aligned with intended message logic.

Integration with downstream systems such as the Editron and HVLP inverters further demonstrated the extensibility of the developed framework, with communication occurring seamlessly over both standard J1939 and H-Protocol CAN. Timing constraints, delta intervals, and signal dependencies were observed in real hardware, strengthening confidence in the robustness of the software logic.

In the following chapter, we shift focus to a detailed evaluation of system performance using metrics derived from the same real-world test data. This includes timing analysis of message cycles, CPU and memory profiling, CAN bus utilization, and frame rate behavior. The chapter also highlights any anomalies encountered and interprets their impact on system stability and responsiveness.

Chapter 7

Results and Analysis

7.1 Chapter Overview

This chapter presents the performance evaluation of the developed CAN communication and visualization framework under simulated and real-world conditions. The analysis covers critical metrics such as data rate distribution, message cycle timing, CPU and memory usage, and frame rendering performance. The goal is to validate the system's ability to meet the real-time communication and visualization requirements of an electrified off-road vehicle using the SAE J1939 protocol.

7.2 Performance Metrics

The CAN bus in the system operates at a nominal data rate of **512 kbps**. As seen in Figure 7.1, the bus utilization reaches approximately **446 kbps**, translating to around 87% usage, which is well within the safe operational threshold. The pie chart illustrates the contribution of each node to the total data load. The Vehicle Control Unit (VCU) is the dominant contributor, accounting for over 40% of the total bandwidth, followed by the Vehicle Interface Gateway (VIG) at roughly 20%. Other components like Editron, PDU, and the HVLP motor controllers contribute smaller but consistent loads. This distribution reflects the control-heavy nature of the VCU and the frequent monitoring messages sent by VIG.

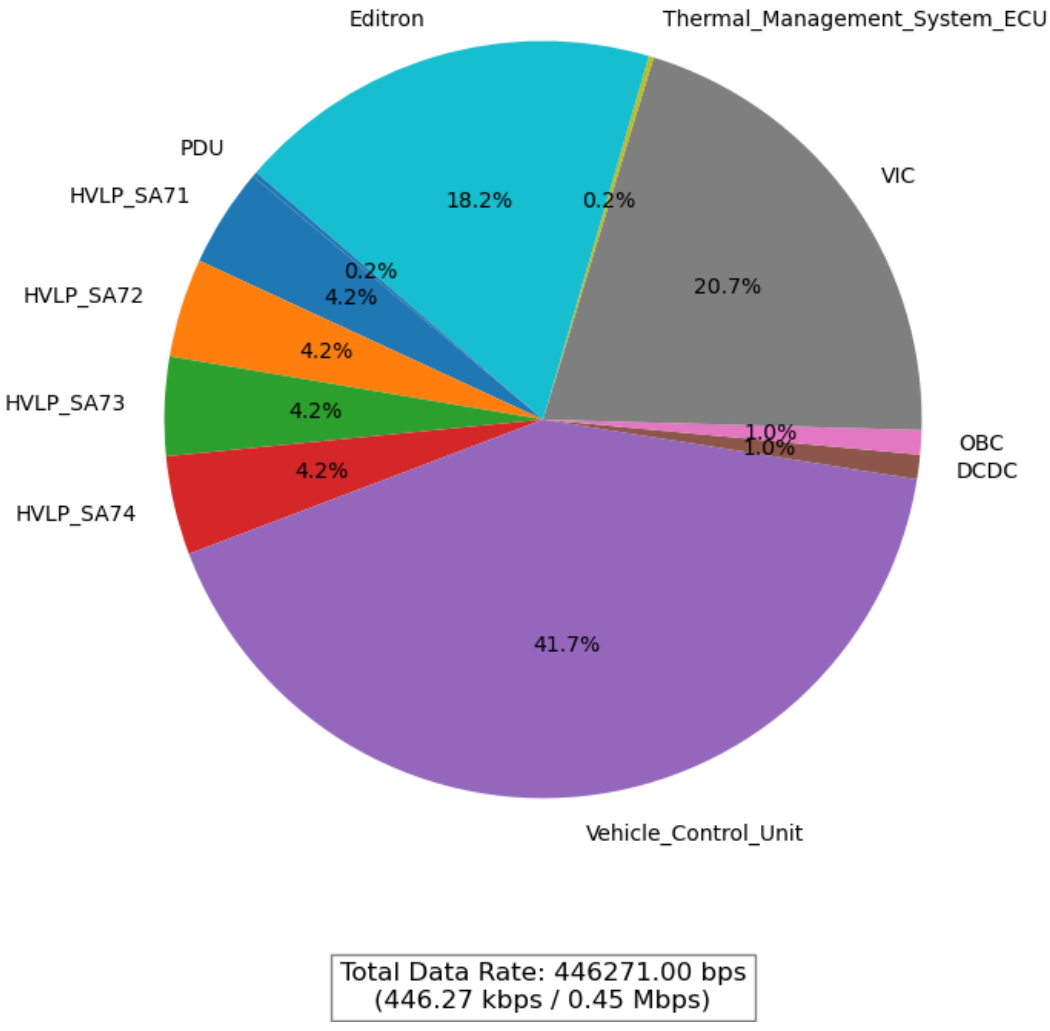

Figure 7.1: Data rate distribution among all transmitting CAN nodes

To evaluate real-time performance, Figure 7.2 presents the comparison of expected vs. actual cycle times for all active CAN messages. Each message is identified by its source CAN ID. The expected cycle time is determined from a predefined DBC or configuration file, while the actual values are computed at runtime from received message timestamps. The graph shows strong consistency across all messages, with minimal deviation from the desired timing. One outlier at 5000 ms corresponds to a

low-priority periodic heartbeat or status message.

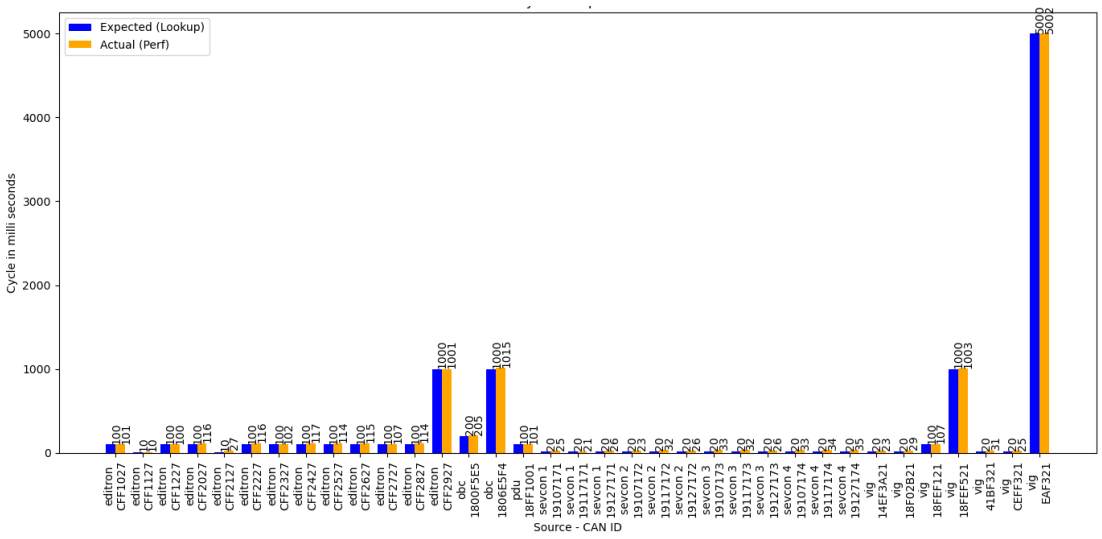

Figure 7.2: Comparison between expected and actual message cycle times

A more focused view is shown in Figure 7.3, which highlights the absolute difference (in milliseconds) between expected and actual cycle times. This helps identify messages with inconsistent timing, potentially caused by scheduling delays or thread contention. The majority of messages show differences between 0 and 17 ms. A few message IDs—like 0xCFF2427 and 0x1806E5F4—consistently show higher offsets, suggesting they could benefit from tighter thread control or prioritized scheduling in future iterations.

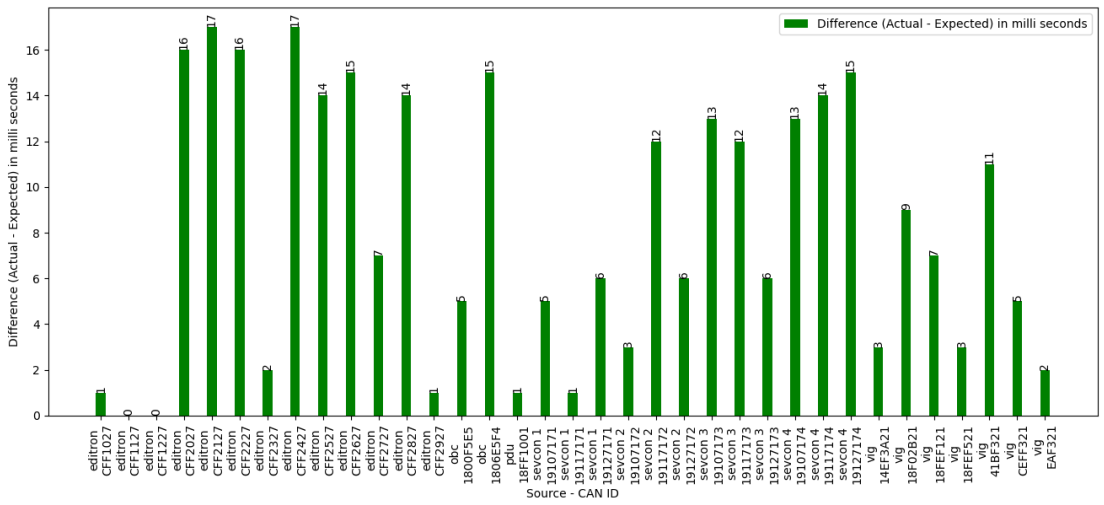

Figure 7.3: Difference between expected and actual message cycle times

Resource efficiency is crucial for embedded deployment. Table 7.1 summarizes CPU and memory metrics logged during extended operation (including active CAN traffic and dashboard visualization). The application consumes approximately 194% CPU (effectively two full logical cores), uses 136 MB of physical memory, and utilizes only 1.7% of total system RAM—well within acceptable limits for an embedded Linux system like Raspberry Pi.

Table 7.1: System Resource Usage

Metric	**Value**	**Description**
CPU Usage	194.0%	Approx. 2 logical cores utilized
Virtual Memory	1.1 GB	Total addressable memory used
Physical Memory	136 MB	Direct memory in use
Shared Memory	64 MB	Used for inter-process sharing
Memory Usage %	1.7%	Of total system memory

The dashboard frame rendering rate is critical for ensuring a responsive user experience during runtime. Figure 7.4 shows the measured average FPS under different

operating scenarios:

- **Without CAN Traffic:** Baseline condition, GUI-only operation.
- **With CAN Traffic:** Standard system with all ECUs active and streaming.
- **With User Interaction + CAN Traffic:** Intensive use case with UI elements actively manipulated while CAN messages are processed.
- **Without User Interaction + CAN Traffic:** Passive monitoring scenario during long runs.
- **With CAN Traffic (3-Hour Load):** Extended runtime test over a continuous 3-hour period with real ECUs, aimed at validating long-term stability and monitoring for performance degradation or thermal throttling. .

The FPS ranges between 32.5 and 34.34 across all scenarios, with minimal variance. This confirms that the Qt-based dashboard remains smooth and highly interactive even under peak load, thanks to efficient use of multithreading and decoupling of UI and backend threads.

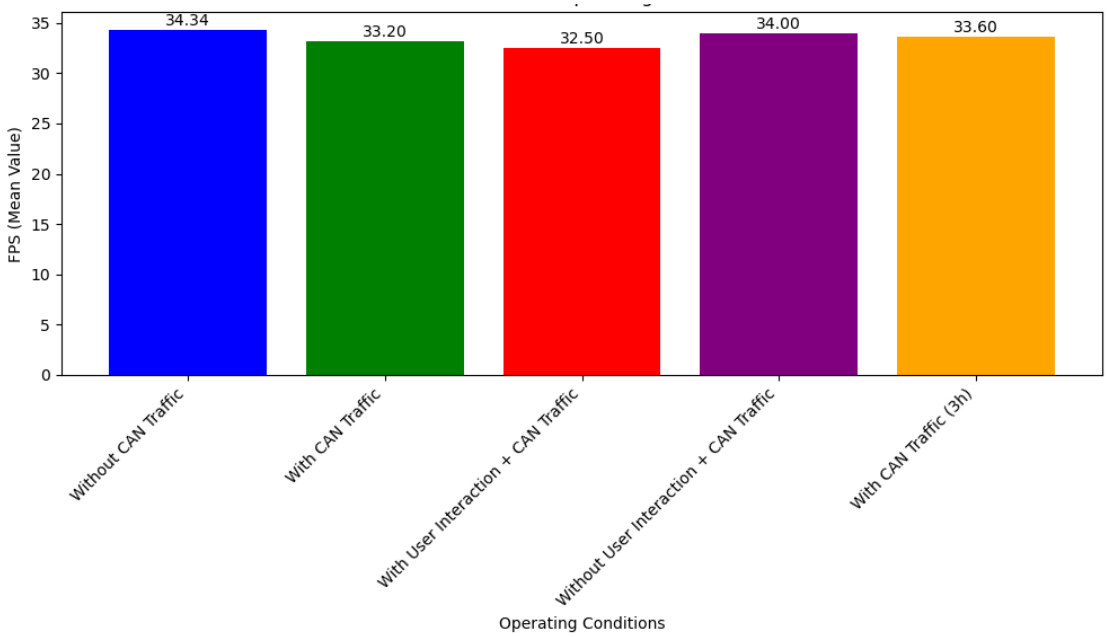

Figure 7.4: FPS across various dashboard operating conditions

7.3 Analysis of Results

The framework achieved strong performance across all evaluated metrics:

- The CAN bus remained well within capacity, with efficient distribution of bandwidth across key nodes.
- Message timing accuracy, with deviations under 10ms, confirmed the reliability of the time-based scheduling logic.
- CPU usage demonstrated that the system could effectively parallelize processing across cores, enabling real-time operation.
- Memory consumption remained minimal, validating the framework's suitability for embedded deployment on platforms such as the Raspberry Pi.
- Real-time visualization maintained smooth and responsive user experience, with high frame rate and low latency even under prolonged and interactive use.

These results demonstrate the effectiveness and reliability of the proposed communication and visualization framework in a real-time, safety-critical electric vehicle environment.

7.4 Chapter Summary

This chapter presented the performance metrics and analysis of the proposed framework under both simulated and real-world conditions. The results demonstrated accurate message cycle timing, efficient CPU and memory utilization, and consistent graphical performance, even under extended runtime and user interaction. These outcomes validate the framework's robustness, responsiveness, and suitability for embedded deployment in electrified off-road vehicles.

The observed performance confirms that the developed communication and visualization system satisfies the core objectives of modularity, reliability, and real-time operation as outlined in the early chapters.

The next chapter concludes the thesis by summarizing the overall contributions, revisiting the core research questions, and outlining potential directions for future work based on the findings presented here.

Chapter 8

Conclusion

8.1 Review of Thesis Content

This thesis addressed the need for a modular and testable communication and visualization framework for electrified off-road vehicles, focusing on systems utilizing the SAE J1939 protocol. The work began by outlining the complexity of coordinating various high-voltage subsystems—such as batteries, inverters, PDUs, and onboard chargers—within a hybrid hydraulic-electric architecture. Emphasis was placed on the critical role of real-time communication, safety checks, and subsystem synchronization in such high-power environments.

A detailed overview of existing tools and frameworks revealed significant limitations in current solutions, including lack of flexibility, high licensing costs, limited J1939 support, and poor integration capabilities for embedded off-road applications. These insights highlighted the need for an open, extensible, and real-time communication framework.

To address this gap, a software architecture was developed using an embedded-friendly Qt-based framework written in C++. It featured components for transmitting and receiving J1939 CAN messages, decoding/encoding based on DBC files, and implementing signal-driven logic via a JSON-based finite state machine. The design prioritized modularity, thread safety, and compatibility with real-time vehicle control systems.

A user-friendly dashboard was implemented to visualize live vehicle data, monitor

control states, and support debugging. It provided dedicated screens for control inputs, signal monitoring, and CAN message inspection, all integrated with the backend communication stack for real-time performance.

The system was first tested in a simulated environment using a hardware-in-the-loop setup, which allowed decoding and encoding of raw J1939 messages, evaluation of fault scenarios, and validation of control sequences before deployment. This approach significantly reduced development time and mitigated integration risks.

Following simulation, the framework was deployed onto the actual prototype vehicle. Real-world integration involved initializing communication with all subsystems, establishing startup/shutdown sequences, and ensuring proper high-voltage coordination. The framework successfully handled charging protocols, safety messaging, and motor control across multiple inverters and ECUs.

Final results confirmed that the framework met its intended goals: enabling standardized, testable communication for electrified off-road vehicles, reducing integration complexity, and supporting traceability and diagnostics through a real-time visualization interface. The solution demonstrated high performance, modular adaptability, and strong alignment with industry protocols like SAE J1939.

8.2 Summary of Research Conclusions

The conclusions drawn from this research are as follows:

- Communication between vehicle subsystems in high-voltage electric off-road vehicles is complex and demands rigorous coordination, safety checks, and traceability—functions that are effectively handled through a structured J1939 protocol framework.

- Existing commercial and open-source solutions lack some combination of key features required for off-road electric platforms: open-source extensibility, embedded optimization, trace analytics, and modular ECU integration.

- The proposed framework fills this gap by offering a highly configurable, testable, and scalable platform designed for embedded electric off-road systems. It integrates advanced FSM logic, visual diagnostics, and real-time CAN traffic parsing

while ensuring compliance with SAE J1939.

- The C++/Qt dashboard enhances development workflows by enabling live data visualization, state monitoring, and debugging, all while maintaining portability and performance suitable for embedded Linux platforms such as the Raspberry Pi.

- Hardware-in-the-loop testing played a critical role in validating the system architecture, identifying issues in CAN communication, and ensuring that contactors, inverters, and charger modules functioned safely before full deployment on the vehicle.

8.3 Recommendations for Future Work

This work opens multiple directions for enhancement and extension of the proposed framework:

- **Full Vehicle-Level Testing:** The framework should be validated on the complete vehicle to assess real-world performance, vibration resilience, and interaction across all subsystems under operational conditions.

- **Advanced Trace and Logging Tools:** Future versions could integrate timeline-based signal plotting, event triggers, and advanced filtering capabilities to better support diagnostics and data analysis during HIL or field testing.

- **CAN FD and Multi-Bus Support:** As next-generation ECUs adopt CAN FD and Ethernet-based communication, the framework can be extended to support high-bandwidth and multi-channel CAN systems.

- **Cybersecurity Integration:** With the growth of connected off-road and autonomous platforms, implementing CAN-layer intrusion detection, message authentication, and fault injection testing is essential for safety.

- **Integration with Autonomy Stacks:** Expanding the framework to interoperate with autonomy platforms like ROS 2 would enable state-aware autonomy modules to react to vehicle-level diagnostics and control states.

- **Automatic DBC Generation:** Development of tools to semi-automatically generate or validate DBC files from known signal databases or XML definitions could reduce manual configuration errors.
- **Modular Plugin System:** A plugin-based extension system could allow rapid deployment of new ECUs, signal handlers, or dashboard views without recompilation.

In conclusion, this thesis delivers a unified and reusable framework that bridges a critical gap in communication and visualization for electrified off-road vehicles. Its open architecture, real-time visual feedback, and robust CAN integration provide a foundation for future research and development in off-road electrification, embedded diagnostics, and vehicle control.

References

[1] Perry Y. Li, Jacob Siefert, and David Bigelow. A hybrid hydraulic-electric architecture (hhea) for high power off-road mobile machines. In *ASME/BATH Symposium on Fluid Power and Motion Control*, volume 59339, page V001T01A011. American Society of Mechanical Engineers, October 2019.

[2] Rahul M. Patil, K. P. Chethan, Rahul Ramaprasad, H. K. Nithin, and Srujan Rangayyan. Infotainment system using can protocol and system on module with qt application for formula-style electric vehicles. In *International Conference on Innovative Computing and Communications: Proceedings of ICICC 2020, Volume 1*, pages 215–226. Springer Singapore, 2021.

[3] David Benedetti, Jacopo Agnelli, Alessio Gagliardi, Pierpaolo Dini, and Sergio Saponara. Design of a digital dashboard on low-cost embedded platform in a fully electric vehicle. In *2020 IEEE International Conference on Environment and Electrical Engineering and 2020 IEEE Industrial and Commercial Power Systems Europe (EEEIC/I&CPS Europe)*, pages 1–5. IEEE, June 2020.

[4] CANoe - ECU & Network Testing. `https://www.vector.com/gb/en/products/products-a-z/software/canoe/#c385766`. Accessed: 2025-03-23.

[5] EPEC - Control System Solutions. `https://www.epectec.com/`. Accessed: 2025-03-23.

[6] MATLAB and Simulink for Electric Vehicle Development. `https://www.mathworks.com/solutions/automotive/electric-vehicle.html`. Accessed: 2025-03-23.

[7] SocketCAN - Controller Area Network. https://docs.kernel.org/networking/can.html. Accessed: 2025-03-23.

[8] Daniel Martensson. Open-SAE-J1939. https://github.com/DanielMartensson/Open-SAE-J1939. Accessed: 2025-03-23.

[9] COVESA (Connected Vehicle Systems Alliance). https://covesa.global/. Accessed: 2025-03-23.

[10] Automotive Grade Linux (AGL). https://www.automotivelinux.org/. Accessed: 2025-03-23.

[11] Eclipse Kuksa. https://projects.eclipse.org/projects/automotive.kuksa. Accessed: 2025-03-23.

[12] Python CAN - CAN/J1939 Tools. https://python-can.readthedocs.io/en/stable/other-tools.html. Accessed: 2025-03-23.

[13] Instrument Control Toolbox - MATLAB. https://www.mathworks.com/products/instrument.html. Accessed: 2025-03-23.

[14] The Yocto Project. *The Yocto Project*, 2025. Available at: https://www.yoctoproject.org [Accessed: 30-Mar-2025].

[15] The Qt Company. *Qt Creator IDE*, 2025. Available at: https://www.qt.io/product/development-tools [Accessed: 30-Mar-2025].

[16] The Qt Company. *Qt CAN Bus API Documentation*, 2025. Available at: https://doc.qt.io/qt-6/qcanbus.html [Accessed: 30-Mar-2025].

[17] The Qt Company. *Qt for Device Creation*, 2025. Available at: https://doc.qt.io/embedded.html [Accessed: 30-Mar-2025].

Appendix A

J1939 Protocol Overview

This appendix provides an overview of the J1939 protocol used in heavy-duty vehicles and outlines key concepts such as message structures, PGNs, SPNs, and the decoding/encoding process using DBC files. It serves as a reference for understanding the CAN communication structure and interpreting raw J1939 data in the context of this thesis.

A.1 Key Features of J1939

A.1.1 What is J1939?

SAE J1939 is a higher-layer protocol built on top of the CAN standard, developed by SAE International. It defines a standardized method for electronic control units (ECUs) in heavy-duty vehicles to communicate with one another over the CAN bus. This common communication framework allows interoperability across vendors, facilitating system integration, diagnostics, and data logging.

While cars often rely on proprietary OEM-specific protocols, the J1939 standard is widely adopted in the heavy-duty automotive domain due to its robustness, extensibility, and compatibility with multi-vendor ECUs.

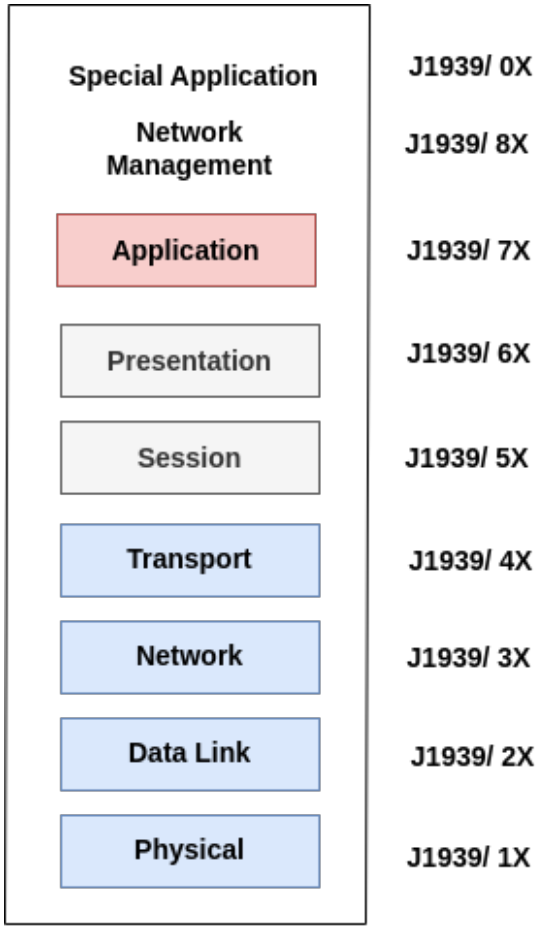

Figure A.1: Illustration of the OSI Model

J1939 operates primarily on the network and transport layers of the OSI model. It extends the capabilities of CAN (which defines the physical and data link layers) by introducing well-defined message formats, identifiers, and data definitions that allow standardized communication across systems.

Other higher-layer CAN protocols include OBD-II, UDS, and CANopen. Among these, J1939 is uniquely tailored to the heavy-duty and off-road industries.

A.2 J1939 Message Structure and Prioritization

J1939 communication is based on 29-bit extended CAN identifiers. These IDs contain embedded fields such as priority, source address, destination address, and the Parameter Group Number (PGN). Messages may carry up to 64 bytes of data, with specific timing and rate requirements defined at the PGN level.

Prioritization is handled via the three most significant bits in the 29-bit CAN ID. Lower numerical values correspond to higher priority, enabling critical messages (e.g., shutdown or fault alerts) to preempt lower-priority data on the bus.

A.3 Parameter Group Numbers (PGNs) and SPNs

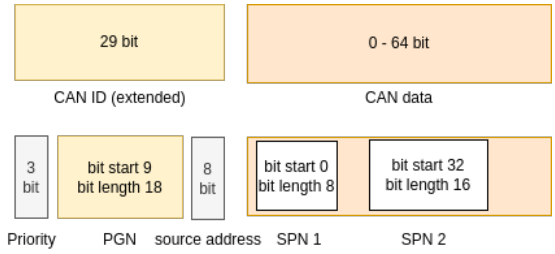

Figure A.2: Parameter Group Number Structure

A PGN (Parameter Group Number) is an 18-bit subset of the CAN ID and serves as the primary identifier for a message in the J1939 protocol. It defines how a CAN frame should be interpreted and which signals (SPNs) are included in the message. A PGN also defines the message's transmission rate, length, and intended use.

Since multiple 29-bit CAN IDs can map to the same PGN (e.g., different source addresses), decoding is performed at the PGN level.

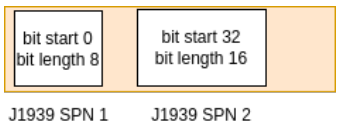

Figure A.3: Suspect Parameter Number (SPN)

SPNs (Suspect Parameter Numbers) represent individual signals within a PGN. Each SPN includes metadata such as:

- Start bit
- Bit length
- Byte order (endianness)
- Scaling factor
- Offset
- Unit (e.g., volts, rpm)

These values are required to decode the raw data payload into meaningful physical values.

A.4 What is a DBC File?

A DBC (Database CAN) file is a standard format used to describe CAN messages, including PGNs and SPNs. It contains definitions for message structure, signal encoding/decoding rules, data lengths, units, and scaling factors. DBC files are commonly used in tools like Vector CANalyzer, Python-CAN, or custom in-house systems to automate CAN data interpretation.

For each PGN, a DBC file lists all associated SPNs and defines how raw binary values should be parsed or constructed.

A.5 How to Decode Raw J1939 Data

Decoding a J1939 message involves converting binary payloads from CAN frames into human-readable physical values using SPN definitions found in a DBC file.

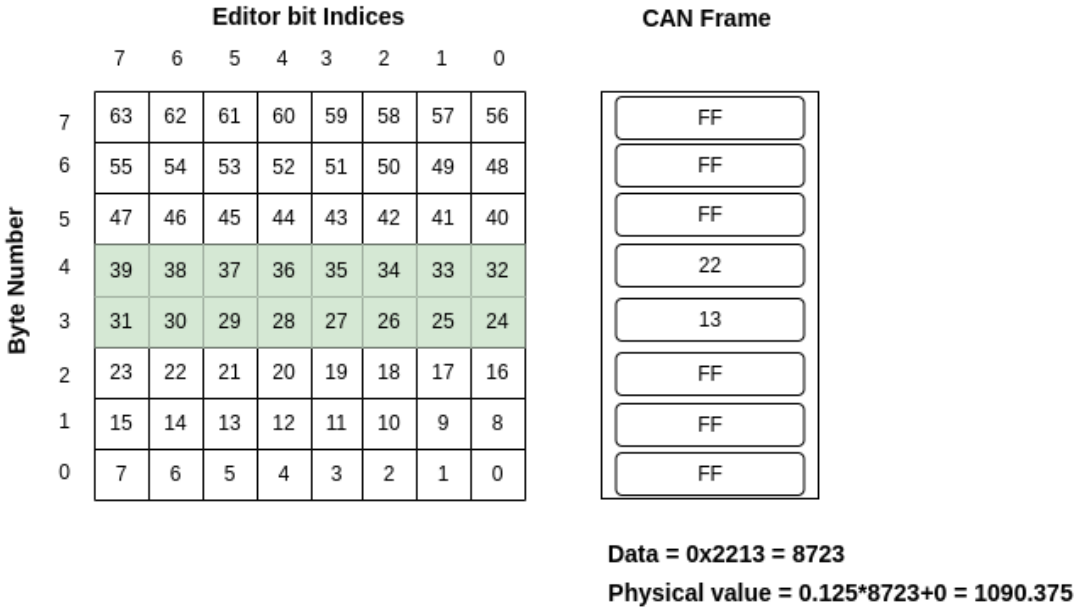

Figure A.4: DBC File Decoding/Encoding

The decoding procedure involves:

1. Extract signal parameters (start bit, length, scale, offset, endianness) from the DBC.

2. Isolate the relevant bits from the CAN payload.

3. Convert to decimal, apply scaling and offset.

   Formula:

   $$\text{Physical Value} = (\text{Decimal Value of Raw Data}) \times \text{Factor} + \text{Offset}$$

4. For example, if a 16-bit signal starts at bit 32 with a scale of 0.125 and offset of 0, and the raw data is 0x0400 (decimal 1024), then:

   $$\text{Physical Value} = 1024 \times 0.125 = 128.0$$

This calculation is typically implemented using a utility function, such as: `decodeSignal(raw_data)`.

A.6 How to Encode J1939 Data into CAN Frames

Encoding J1939 messages is the reverse process: given a physical value, convert it into raw binary data using signal definitions and insert it into the appropriate bit field in the CAN frame.

1. Retrieve SPN details from the DBC file (start bit, length, scale, offset, endianness).
2. Calculate the raw integer value.

   Formula:

   $$\text{Raw Value} = \frac{(\text{Physical Value} - \text{Offset})}{\text{Factor}}$$

3. Insert this value into the correct location in the message payload.

The encoder logic is typically implemented using a function such as: `encodeSignal-(physical_value)`.

This is particularly useful when crafting custom control messages from the VCU or simulating signal injections during HIL testing.

A.7 Safety Header and Safety Data Messages

The SAE J1939-76 protocol introduces an additional safety mechanism that enhances the integrity and predictability of message exchanges between electronic control units (ECUs) in high-voltage automotive systems. This mechanism involves the transmission of a **Safety Header Message (SHM)** immediately before the corresponding **Safety Data Message (SDM)**.

A.7.1 Safety Header Message (SHM)

The Safety Header Message is a lightweight message used to precede safety-critical CAN messages. It serves two primary purposes:

- To notify the receiver that a Safety Data Message is imminent
- To validate the integrity of the upcoming data using a checksum

The SHM includes metadata such as the PGN of the target SDM and a 32-bit cyclic redundancy check (CRC) calculated from the SDM payload using a predefined polynomial (e.g., 0x6938392D). This ensures that any manipulation or corruption of the data in transit can be detected before use.

In the system described in this thesis, the SHM is **specific to the Vehicle Interface Gateway (VIG)**, as the VIG adheres strictly to the J1939-76 safety layer. The SHM must be transmitted at its defined cycle time, and any timing mismatches or checksum failures result in the VIG rejecting the associated SDM.

A.7.2 Safety Data Message (SDM)

The Safety Data Message is the standard J1939 message that contains the actual signal data, such as voltage limits, charging commands, system states, or control requests. In systems that comply with J1939-76, the SDM must always be sent immediately after the corresponding SHM.

While the SHM is VIG-specific, SDMs are used more broadly across all CAN nodes. For example, the PDU, battery system, and inverter ECUs also rely on safety-related PGNs, but do not necessarily require a preceding SHM unless the node enforces J1939-76.

A.7.3 Message Pairing and Timing Requirements

The pairing of SHM and SDM is subject to strict timing constraints. The SDM must follow the SHM within a defined delta time (e.g., within a few milliseconds), and both messages must be sent at their configured cycle times to maintain compliance. Missed or out-of-order transmissions can lead to faults or ignored messages at the receiving ECU.

A.7.4 Implementation Details

In this project, both SHM and SDM are generated using dedicated encoding functions (`CreateSHMData()` and `CreateSDMData()`) within the embedded software. The data for the SDM is first populated and used to compute the SHM checksum. Both messages are then queued for transmission based on their assigned cycle times.

Appendix B

Supplementary Resources

For access to the complete source code, implementation details, and supplementary materials related to this thesis, please refer to the following resources:

- **GitHub Repository:**
  `https://github.com/Sujeendra/ev-2025`
- **Documentation and Dashboard Walkthrough:**
  `https://ev-2025.web.app/`

These resources include all the code for the Vehicle Control Unit (VCU), dashboard UI, CAN communication framework, and system state machine logic. The documentation site provides real-time demos, visual explanations, and step-by-step setup instructions.

Appendix C

VCU Software: Skeleton Code

C.1 C++ Backend Code

This appendix provides a high-level skeleton of the VCU application written in C++. The core runtime includes initialization of CAN interfaces, loading of DBC configurations, setup of shared memory (`dataVector`), and execution of key threads including the CAN sender, receiver, and finite state machine logic. The code structure shown below is simplified for readability, emphasizing the modularity and memory-centered control strategy discussed in Chapter 3.

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QDateTime>
#include <QTimeZone>
#include <QObject>
#include <QQmlContext>
#include <QtVirtualKeyboard>
#include <QTimer>
#include <qqml.h>
#include <QAbstractTableModel>
#include <QAbstractItemModel>
#include <QVariant>
#include <QCanBus>
#include <QCanDbcFileParser>
#include <QCanBusFrame>
#include <QCanBusDevice>
```

```
17 #include <QRandomGenerator>
18 #include <cmath>
19 #include <QtSql/QtSql>
20 #include <QDebug>
21 #include "autogen/environment.h"
22 #include <algorithm>
23 #include <vector>
24 #include <gpiod.h>
25 #include <QProcess>
26 #include <QNetworkInterface>
27 #include <thread>
28 #include "SystemManager.h"
29 #include "LogManager.h"
30 #include "WifiManager.h"
31 #include "Crc32Calculator.h"
32 #include "DatabaseManager.h"
33 #include "SignalValueMap.h"
34 #include "utility.h"
35 #include "TreeModel.h"
36 #include "TimeManager.h"
37 #include "fa.h"
38 #include <QTextStream>
39 #include <QHash>
40 #include <QFile>
41 #include <QJsonDocument>
42 #include <QJsonObject>
43 #include <QJsonArray>
44 #include <QJsonValue>
45 #include <QJsonParseError>
46
47 bool IsConnectForDriving = false;
48 bool IsDataTransmissionStarted = false;
49 bool IsConnectedForDCCharging = false;
50 bool IsConnectedForACCharging = false;
51 LogManager logManager;
52 bool SendGPIOSignal = false;
53 bool HVReq = false; // High voltage request
54 bool IsBatteryContactorClosed = false;
```

```
55 std::vector<uint32_t> frameIdsToForward = {0x1918FF71, 0x1919FF71, 0
       x191AFF71, 0xCFF91FD, 0xCFF92FD};
56 QMap<QString, double> dataVec;
57
58 //   <-----FSM related UTILS Starts here -------->
59
60 QJsonObject loadFSMFromFile(const QString &path)
61 {
62     QFile file(path);
63     if (!file.open(QIODevice::ReadOnly))
64     {
65         qWarning() << "Failed to open FSM file";
66         return {};
67     }
68     QByteArray data = file.readAll();
69     QJsonParseError err;
70     QJsonDocument doc = QJsonDocument::fromJson(data, &err);
71     if (err.error != QJsonParseError::NoError)
72     {
73         qWarning() << "JSON parse error:" << err.errorString();
74         return {};
75     }
76     return doc.object();
77 }
78
79 bool evaluateSingleCondition(const QJsonObject &cond,
80                              const QMap<QString, SignalInfo> &
                                    signalValueMap)
81 {
82     QString key = cond["key"].toString();
83     if (!signalValueMap.contains(key))
84         return false;
85
86     double actual = signalValueMap[key].value;
87
88     if (cond.contains("eq"))
89         return qFuzzyCompare(actual + 1.0, cond["eq"].toDouble() + 1.0);
90     if (cond.contains("ne"))
91         return !qFuzzyCompare(actual + 1.0, cond["ne"].toDouble() + 1.0);
```

```
92      if (cond.contains("lt"))
93          return actual < cond["lt"].toDouble();
94      if (cond.contains("gt"))
95          return actual > cond["gt"].toDouble();
96      if (cond.contains("le"))
97          return actual <= cond["le"].toDouble();
98      if (cond.contains("ge"))
99          return actual >= cond["ge"].toDouble();
100
101     return false;
102 }
103
104 bool evaluateFSMCondition(const QJsonObject &condition,
105                           const QMap<QString, SignalInfo> &signalValueMap
                              )
106 {
107     if (condition.contains("all"))
108     {
109         QJsonArray all = condition["all"].toArray();
110         for (const auto &item : all)
111         {
112             if (!evaluateSingleCondition(item.toObject(), signalValueMap)
                    )
113                 return false;
114         }
115         return true;
116     }
117     if (condition.contains("any"))
118     {
119         QJsonArray any = condition["any"].toArray();
120         for (const auto &item : any)
121         {
122             if (evaluateSingleCondition(item.toObject(), signalValueMap))
123                 return true;
124         }
125         return false;
126     }
127
128     // fallback for old flat condition style:
```

```
129     for (auto it = condition.begin(); it != condition.end(); ++it)
130     {
131         const QString &key = it.key();
132         double expected = it.value().toDouble();
133         if (!signalValueMap.contains(key))
134             return false;
135         if (!qFuzzyCompare(signalValueMap[key].value + 1.0, expected +
                1.0))
136             return false;
137     }
138     return true;
139 }
140 
141 // recheck if the new key created works without explicitly inserting
142 void performFSMAction(const QJsonObject &action,
143                       QMap<QString, double> &dataVec)
144 {
145     QString key = action["signal"].toString().trimmed();
146     double value = action["value"].toDouble();
147     dataVec[key] = value;
148     qDebug() << "[FSM Action] Set" << key << "=" << value;
149 }
150 
151 QString runFSMTransition(QJsonObject &fsm,
152                          const QString &currentState,
153                          const QMap<QString, SignalInfo> &signalValueMap,
154                          QMap<QString, double> &dataVec)
155 {
156     QJsonArray states = fsm["states"].toArray();
157 
158     for (const QJsonValue &stateVal : states)
159     {
160         QJsonObject stateObj = stateVal.toObject();
161         if (stateObj["name"].toString() != currentState)
162             continue;
163 
164         QJsonArray transitions = stateObj["transitions"].toArray();
165         for (const QJsonValue &transVal : transitions)
166         {
```

```
167             QJsonObject trans = transVal.toObject();
168             if (evaluateFSMCondition(trans["condition"].toObject(),
                    signalValueMap))
169             {
170                 QJsonArray actions = trans["actions"].toArray();
171                 for (const QJsonValue &actVal : actions)
172                 {
173                     performFSMAction(actVal.toObject(), dataVec);
174                 }
175                 return trans["nextState"].toString();
176             }
177         }
178     }
179
180     return currentState; // No transition
181 }
182 //    <------FSM related Utils ends here --------------->
183
184 // can 0 is always code as vehicle CAN
185
186 class TableModel : public QAbstractTableModel
187 {
188     Q_OBJECT
189     QML_ELEMENT
190
191 public:
192     int rowCount(const QModelIndex & = QModelIndex()) const override
193     {
194         return 200;
195     }
196
197     int columnCount(const QModelIndex & = QModelIndex()) const override
198     {
199         return 200;
200     }
201
202     QVariant data(const QModelIndex &index, int role) const override
203     {
204         switch (role)
```

```
205         {
206         case Qt::DisplayRole:
207             return QString("%1, %2").arg(index.column()).arg(index.row())
                    ;
208         default:
209             break;
210         }
211 
212         return QVariant();
213     }
214 
215     QHash<int, QByteArray> roleNames() const override
216     {
217         return {{Qt::DisplayRole, "display"}};
218     }
219 };
220 // Function to find and return a Message object by matching the first SDM
        entry's message name
221 Message *findMessageInSDM(const std::string &messageName, std::
        unordered_map<std::string, Message> &messages)
222 {
223     // Iterate through each message in the map
224     for (auto &[pgn, message] : messages)
225     {
226         // Check if the SDM vector has entries and if the first entry's
                message name matches
227         if (!message.SDM.empty() && message.SDM[0].message == messageName
                )
228         {
229             return &message; // Return the matching message object
230         }
231     }
232     // If no match was found, return an first element
233     return nullptr;
234 }
235 
236 void CreateSDMData(QCanDbcFileParser &fileParser, std::unordered_map<std
        ::string, Message> &messages)
237 {
```

```
238     std::string target_node = "Vehicle_Control_Unit";
239
240     // Loop over all message descriptions from the parsed DBC file
241     const auto descriptions = fileParser.messageDescriptions();
242     for (const auto &description : descriptions)
243     {
244         if (description.transmitter() == target_node)
245         {
246             // Find the matching message in SDM
247             Message *result = findMessageInSDM(description.name().
                    toStdString(), messages);
248             if (result != nullptr)
249             {
250
251                 // Loop over all signal descriptions in the message
252                 for (const auto &signal : description.signalDescriptions
                        ())
253                 {
254                     auto value_it = std::find_if((*result).SDM.begin(),
                            (*result).SDM.end(),
255                                                  [&signal](const
                                                      DataEntry &entry)
256                                                  {
257                                                      return entry.signal
                                                          == signal.name()
                                                          .toStdString();
258                                                  });
259
260                     // Check if the element was found in SDM
261                     if (value_it != (*result).SDM.end())
262                     {
263                         QString key = QString::fromStdString(target_node)
                                + "." +
264                                       description.name() + "." +
265                                       signal.name();
266                         // Add check to read value from shared data if it
                                has changed there
267                         for (int i = 0; i < dataVec.size(); ++i)
268                         {
```

```
269                             if (dataVec.contains(key))
270                             {
271                                 value_it->value = dataVec[key];
272                                 break;
273                             }
274                         }
275
276                         double value = value_it->value;
277                         // Encode the signal value using the QtCanDbc
                                signal encoding method
278                         encodeSignal(value, signal.offset(), signal.
                                factor(), signal.startBit(), signal.bitLength
                                (), (*result).sdmSignedData, signal.
                                dataEndian(), signal.dataFormat());
279                     }
280                 }
281             }
282         }
283     }
284 }
285 class TimeDateResponder : public QObject
286 {
287     Q_OBJECT
288
289 public:
290     explicit TimeDateResponder(QMap<QString, SignalInfo> &signalValueMap,
            QJsonObject &fsm)
291         : signalValueMap(signalValueMap), fsm(fsm) {}
292
293     void initialize()
294     {
295         // Initialize CAN Bus Device
296         canDevice = QCanBus::instance()->createDevice("socketcan", "can0"
                );
297         if (!canDevice)
298         {
299             logManager.addLog("Error: Could not create CAN device.");
300             return;
301         }
```

```
302
303         if (!canDevice->connectDevice())
304         {
305             logManager.addLog("Error: Could not connect to CAN device.");
306             return;
307         }
308
309         logManager.addLog("CAN 0 device connected successfully.");
310         // Initialize CAN1
311         canDevice1 = QCanBus::instance()->createDevice("socketcan", "can1
                ");
312         if (!canDevice1)
313         {
314             logManager.addLog("Error: Could not create CAN1 device.");
315             return;
316         }
317         if (!canDevice1->connectDevice())
318         {
319             logManager.addLog("Error: Could not connect to CAN1.");
320             return;
321         }
322         logManager.addLog("CAN1 connected successfully.");
323         // Connect to frame reception
324         connect(canDevice, &QCanBusDevice::framesReceived, this, &
                TimeDateResponder::processFrames);
325     }
326
327 private:
328     QCanBusDevice *canDevice = nullptr;
329     QCanBusDevice *canDevice1 = nullptr;
330     QMap<QString, SignalInfo> &signalValueMap;
331     QJsonObject &fsm;
332
333     void processFrames()
334     {
335
336         QString currentState = fsm["initialState"].toString();
337         qDebug() << "Starting FSM in state:" << currentState;
338         while (canDevice->framesAvailable())
```

```
339         {
340             QCanBusFrame frame = canDevice->readFrame();
341 
342             if (isTimeDateRequest(frame))
343             {
344                 // Prepare response frame
345                 QCanBusFrame responseFrame = createTimeDateResponse();
346 
347                 // Send response
348                 canDevice->writeFrame(responseFrame);
349             }
350 
351             QString nextState = runFSMTransition(fsm, currentState,
                    signalValueMap, dataVec);
352             if (nextState != currentState)
353             {
354                 qDebug() << "[FSM] Transitioned:" << currentState << "to"
                        << nextState;
355                 currentState = nextState;
356             }
357             else
358             {
359                 qDebug() << "[FSM] No transition from state:" <<
                        currentState;
360             }
361             // frame forward to CAN 1
362             if (std::find(frameIdsToForward.begin(), frameIdsToForward.
                    end(), frame.frameId()) != frameIdsToForward.end())
363             {
364                 // Forward frame to can1
365                 canDevice1->writeFrame(frame);
366             }
367         }
368     }
369 
370     bool isTimeDateRequest(const QCanBusFrame &frame)
371     {
372         return (frame.frameId()) == 0xEA21F3 &&
373                frame.payload()[0] == 0xE6 &&
```

```
374                 frame.payload()[1] == 0xFE &&
375                 frame.payload()[2] == 0x00;
376     }
377
378     QCanBusFrame createTimeDateResponse()
379     {
380         QCanBusFrame frame;
381         frame.setFrameId(0x18FEE621);
382         frame.setFrameType(QCanBusFrame::DataFrame);
383         QByteArray payload(8, 0);
384         // Get the current time in UTC
385         QDateTime utcTime = QDateTime::currentDateTimeUtc();
386
387         // Populate UTC time
388         payload[0] = utcTime.time().second() / 0.25;
389         payload[1] = utcTime.time().minute();
390         payload[2] = utcTime.time().hour();
391         payload[3] = utcTime.date().month();
392         payload[4] = utcTime.date().day() / 0.25;
393         payload[5] = utcTime.date().year() - 1985;
394
395         // Set offset to 0
396         payload[6] = 125;
397         payload[7] = 125;
398         frame.setPayload(payload);
399         return frame;
400     }
401 };
402
403 void printModelData(QSqlQueryModel *model)
404 {
405     // Check if the model is valid
406     if (model && model->rowCount() > 0)
407     {
408         // Loop through all the rows
409         for (int row = 0; row < model->rowCount(); ++row)
410         {
411             // Loop through all columns
412             QString rowData;
```

```
413             for (int col = 0; col < model->columnCount(); ++col)
414             {
415                 rowData += model->data(model->index(row, col)).toString()
                        + " | ";
416             }
417             qDebug() << rowData; // Print each row
418         }
419     }
420     else
421     {
422         qDebug() << "No data available in the model";
423     }
424 }
425
426 void CreateSHMData(std::pair<const std::string, Message> &pair, uint8_t
    decimalValue)
427 {
428     auto &pgn = pair.first;
429     auto &message = pair.second;
430
431     auto hex29bit = message.messageId29Bit;
432     hex29bit &= 0x1FFFFFFF;
433
434     // Extract 8 bits from the least significant side (right-hand side)
435     uint8_t first8 = hex29bit & 0xFF;         // Extract bits 0-7 source
        address
436     uint8_t second8 = (hex29bit >> 8) & 0xFF; // Extract bits 8-15 pdu-s
437     uint8_t third8 = (hex29bit >> 16) & 0xFF; // Extract bits 16-23 pdu-f
438
439     // Negate (bitwise NOT) each 8-bit section
440     first8 = ~first8;
441     second8 = ~second8;
442     third8 = ~third8;
443
444     uint8_t maskedValue = decimalValue & 0x1F; // 0x1F is binary 00011111
445     // Shift the masked 5-bit value to the left by 3 to make room for the
         last 3 bits
446     uint8_t result = (maskedValue << 3) | 0x07; // 0x07 is binary
        00000111
```

```
447
448     uint32_t high = combineBytesTo32Bit(result, first8, second8, third8);
449
450     // Input bytes as specified in the question
451     // need to be carefull because the data might have negative data --
            need to change the code here
452     std::array<uint8_t, 8> inputBytes = message.sdmSignedData; // data to
            be loaded from excel too specifically formed
453
454     Crc32Calculator calculator;
455     calculator.CalculateCrcTable_CRC32();
456     uint32_t low = ReverseByteOrder(calculator.Compute_CRC32(inputBytes))
            ;
457
458     uint64_t combined = CombineTo64Bit(high, low);
459     for (int i = 7; i >= 0; i--)
460     {
461         message.shmSignedData[i] = static_cast<uint8_t>((combined >> (8 *
                (7 - i))) & 0xFF);
462     }
463 }
464 void delayedSetGpio(int pin, bool status)
465 {
466     std::this_thread::sleep_for(std::chrono::milliseconds(2000)); //
            Sleep for 2 seconds
467     setGpioPin(pin, status);                                        // Call
            the function after sleep
468 }
469 class CanMessageSenderGroup1 : public QThread
470 {
471     Q_OBJECT
472 public:
473     CanMessageSenderGroup1(QCanDbcFileParser &parser, std::unordered_map<
            std::string, Message> &messages)
474         : fileParser(parser), messages(messages) {}
475
476 protected:
477     void run() override
478     {
```

```
479         sendShmSDMMessage(fileParser, messages);
480     }
481
482 private:
483     QCanDbcFileParser &fileParser;
484     std::unordered_map<std::string, Message> &messages;
485
486     void sendShmSDMMessage(QCanDbcFileParser &fileParser, std::
            unordered_map<std::string, Message> &messages)
487     {
488
489         std::unordered_map<std::string, std::chrono::steady_clock::
                time_point> nextSendTime;
490         for (auto &pair : messages)
491         {
492             nextSendTime[pair.first] = std::chrono::steady_clock::now();
493         }
494
495         QCanBusDevice *canBusDevice = QCanBus::instance()->createDevice("
                socketcan", "can0");
496         canBusDevice->connectDevice();
497         bool localGPIODataSentAlreadyfor1 = false;
498         bool localGPIODataSentAlreadyfor0 = true;
499         while (true)
500         {
501
502             if (!IsDataTransmissionStarted)
503             {
504                 // reset timing
505                 for (auto &pair : messages)
506                 {
507                     nextSendTime[pair.first] = std::chrono::steady_clock
                            ::now();
508                 }
509                 continue;
510             }
511
512             // Prepare SHM and SDM data and send them via CAN bus
513             for (auto &pair : messages)
```

```
514             {
515
516                 auto &message = pair.second;
517                 // if not connected for driving dont send this message
518                 if (message.SDM[0].message == "S1_ES_PGN61427" && !
                        IsConnectForDriving && !HVReq)
519                 {
520                     nextSendTime[pair.first] = std::chrono::steady_clock
                            ::now();
521                     continue;
522                 }
523
524                 std::string messageId = pair.first;
525                 // if (message.sheetName != "Main")
526                 //     continue;
527
528                 auto now = std::chrono::steady_clock::now();
529                 if (now >= nextSendTime[messageId])
530                 {
531                     auto timeDifference = std::chrono::duration_cast<std
                            ::chrono::milliseconds>(now - nextSendTime[
                            messageId]);
532                     // Add the cycleTime to the timeDifference
533                     auto totalTime = timeDifference + std::chrono::
                            milliseconds(static_cast<int>(message.SDM[0].
                            cycleTime));
534
535                     // stream << totalTime.count() << ","
536                     //        <<  QString::number(message.messageId29Bit,
                             16).toUpper() << "\n";
537                     CreateSDMData(fileParser, messages);
538
539                     if (message.sheetName == "Main")
540                         CreateSHMData(pair, message.shmCounter);
541
542                     if (message.sheetName == "Main")
543                     {
544                         std::string target_node = "Vehicle_Control_Unit";
```

```
545                     for (const auto &msg : fileParser.
                            messageDescriptions())
546                     {
547                         if (msg.transmitter() == target_node &&
                                message.SHM[0].message == msg.name())
548                         {
549
550                             quint32 shmId = static_cast<quint32>(msg.
                                    uniqueId());
551                             QByteArray shmDataArray(reinterpret_cast<
                                    const char *>(message.shmSignedData.
                                    data()), msg.size());
552                             QCanBusFrame shmFrame(shmId, shmDataArray
                                    );
553                             shmFrame.setFrameType(QCanBusFrame::
                                    DataFrame);
554                             // if(msg.name()!="S1_ES_PGN61427_SHM_Rx
                                    ")
555                             canBusDevice->writeFrame(shmFrame);
556
557                             nextSendTime[messageId] += std::chrono::
                                    milliseconds(static_cast<int>(message
                                    .SHM[0].cycleTime));
558
559                             QThread::msleep(static_cast<unsigned long
                                    >(message.SDM[0].deltaTime));
560
561                             QCanBusFrame sdmFrame(message.
                                    messageId29Bit, QByteArray(
                                    reinterpret_cast<const char *>(
                                    message.sdmSignedData.data()), msg.
                                    size()));
562                             sdmFrame.setFrameType(QCanBusFrame::
                                    DataFrame);
563                             canBusDevice->writeFrame(sdmFrame);
564                             message.incrementCounter();
565                             break;
566                         }
567                     }
```

```
568                     }
569                     else
570                     {
571                         // version 2 changes using dlc from the dbc to
                                send only data in the payload
572                         std::string target_node = "Vehicle_Control_Unit";
573                         for (const auto &msg : fileParser.
                                messageDescriptions())
574                         {
575                             if (msg.transmitter() == target_node &&
                                    message.SDM[0].message == msg.name())
576                             {
577                                 nextSendTime[messageId] += std::chrono::
                                        milliseconds(static_cast<int>(message
                                        .SDM[0].cycleTime));
578                                 QCanBusFrame sdmFrame(message.
                                        messageId29Bit, QByteArray(
                                        reinterpret_cast<const char *>(
                                        message.sdmSignedData.data()), msg.
                                        size()));
579                                 sdmFrame.setFrameType(QCanBusFrame::
                                        DataFrame);
580                                 canBusDevice->writeFrame(sdmFrame);
581                                 break;
582                             }
583                         }
584                     }
585                 }
586             }
587             // send this after safety signals are sent
588             if (!localGPIODataSentAlreadyfor0 && SendGPIOSignal)
589             {
590
591                 localGPIODataSentAlreadyfor0 = true;
592                 localGPIODataSentAlreadyfor1 = false;
593                 SendGPIOSignal = false;
594
595                 std::thread t([]()
596                               { delayedSetGpio(17, false); });
```

```
597
598                 // Detach the thread so it runs independently
599                 t.detach();
600             }
601             else if (!localGPIODataSentAlreadyfor1 && SendGPIOSignal)
602             {
603                 localGPIODataSentAlreadyfor1 = true;
604                 localGPIODataSentAlreadyfor0 = false;
605                 SendGPIOSignal = false;
606
607                 // Start a new thread and pass arguments using lambda
608                 std::thread t([]()
609                               { delayedSetGpio(17, true); });
610
611                 // Detach the thread so it runs independently
612                 t.detach();
613             }
614         }
615     }
616 };
617
618 class CanMessageSenderGroup2 : public QThread
619 {
620     Q_OBJECT
621 public:
622     CanMessageSenderGroup2(QCanDbcFileParser &parser, std::unordered_map<
        std::string, Message> &messages)
623         : fileParser(parser), messages(messages) {}
624
625 protected:
626     void run() override
627     {
628         sendShmSDMMessage(fileParser, messages);
629     }
630
631 private:
632     QCanDbcFileParser &fileParser;
633     std::unordered_map<std::string, Message> &messages;
634
```

```
635     void sendShmSDMMessage(QCanDbcFileParser &fileParser, std::
            unordered_map<std::string, Message> &messages)
636     {
637
638         std::unordered_map<std::string, std::chrono::steady_clock::
                time_point> nextSendTime;
639         for (auto &pair : messages)
640         {
641             nextSendTime[pair.first] = std::chrono::steady_clock::now();
642         }
643
644         QCanBusDevice *canBusDevice = QCanBus::instance()->createDevice("
                socketcan", "can0");
645         canBusDevice->connectDevice();
646         while (true)
647         {
648
649             if (!IsDataTransmissionStarted)
650             {
651                 // reset timing
652                 for (auto &pair : messages)
653                 {
654                     nextSendTime[pair.first] = std::chrono::steady_clock
                            ::now();
655                 }
656                 continue;
657             }
658
659             // Prepare SHM and SDM data and send them via CAN bus
660             for (auto &pair : messages)
661             {
662
663                 auto &message = pair.second;
664                 // if not connected for driving dont send this message
665                 if (message.SDM[0].message == "S1_ES_PGN61427" && !
                        IsConnectForDriving && !HVReq)
666                 {
667                     nextSendTime[pair.first] = std::chrono::steady_clock
                            ::now();
```

```
668                     continue;
669                 }
670
671                 std::string messageId = pair.first;
672                 // if (message.sheetName != "Main")
673                 //     continue;
674
675                 auto now = std::chrono::steady_clock::now();
676                 if (now >= nextSendTime[messageId])
677                 {
678                     auto timeDifference = std::chrono::duration_cast<std
                        ::chrono::milliseconds>(now - nextSendTime[
                        messageId]);
679                     // Add the cycleTime to the timeDifference
680                     auto totalTime = timeDifference + std::chrono::
                        milliseconds(static_cast<int>(message.SDM[0].
                        cycleTime));
681
682                     // stream << totalTime.count() << ","
683                     //        <<  QString::number(message.messageId29Bit,
                         16).toUpper() << "\n";
684                     CreateSDMData(fileParser, messages);
685
686                     if (message.sheetName == "Main")
687                         CreateSHMData(pair, message.shmCounter);
688
689                     if (message.sheetName == "Main")
690                     {
691                         std::string target_node = "Vehicle_Control_Unit";
692                         for (const auto &msg : fileParser.
                            messageDescriptions())
693                         {
694                             if (msg.transmitter() == target_node &&
                                message.SHM[0].message == msg.name())
695                             {
696
697                                 quint32 shmId = static_cast<quint32>(msg.
                                    uniqueId());
```

```
698                         QByteArray shmDataArray(reinterpret_cast<
                                const char *>(message.shmSignedData.
                                data()), msg.size());
699                         QCanBusFrame shmFrame(shmId, shmDataArray
                                );
700                         shmFrame.setFrameType(QCanBusFrame::
                                DataFrame);
701                         // if(msg.name()!="S1_ES_PGN61427_SHM_Rx
                                ")
702                         canBusDevice->writeFrame(shmFrame);
703
704                         nextSendTime[messageId] += std::chrono::
                                milliseconds(static_cast<int>(message
                                .SHM[0].cycleTime));
705
706                         QThread::msleep(static_cast<unsigned long
                                >(message.SDM[0].deltaTime));
707
708                         QCanBusFrame sdmFrame(message.
                                messageId29Bit, QByteArray(
                                reinterpret_cast<const char *>(
                                message.sdmSignedData.data()), msg.
                                size()));
709                         sdmFrame.setFrameType(QCanBusFrame::
                                DataFrame);
710                         canBusDevice->writeFrame(sdmFrame);
711                         message.incrementCounter();
712                         break;
713                     }
714                 }
715             }
716             else
717             {
718                 // version 2 changes using dlc from the dbc to
                        send only data in the payload
719                 std::string target_node = "Vehicle_Control_Unit";
720                 for (const auto &msg : fileParser.
                        messageDescriptions())
721                 {
```

```
722                             if (msg.transmitter() == target_node &&
                                    message.SDM[0].message == msg.name())
723                             {
724                                 nextSendTime[messageId] += std::chrono::
                                        milliseconds(static_cast<int>(message
                                        .SDM[0].cycleTime));
725                                 QCanBusFrame sdmFrame(message.
                                        messageId29Bit, QByteArray(
                                        reinterpret_cast<const char *>(
                                        message.sdmSignedData.data()), msg.
                                        size()));
726                                 sdmFrame.setFrameType(QCanBusFrame::
                                        DataFrame);
727                                 canBusDevice->writeFrame(sdmFrame);
728                                 break;
729                             }
730                         }
731                     }
732                 }
733             }
734         }
735     }
736 };
737
738 class MainController : public QObject
739 {
740     Q_OBJECT
741
742 public:
743     explicit MainController(QObject *parent = nullptr) : QObject(parent)
            {}
744
745     // Make the function invokable using Q_INVOKABLE
746     Q_INVOKABLE void toggleStartStop(bool checked)
747     {
748         if (checked)
749         {
750             IsDataTransmissionStarted = true;
```

```
751             QDateTime cstTime = QDateTime::currentDateTimeUtc().
                    toTimeZone(QTimeZone("America/Chicago"));
752             logManager.addLog("Vehicle CAN Transmission Started for
                    Battery initilisation at: " + cstTime.toString());
753         }
754         else
755         {
756             IsDataTransmissionStarted = false;
757             HVReq = false;
758         }
759         SendGPIOSignal = true;
760     }
761
762     // possibly check if these keys exists before accesing to avoid
        crashes in the UI
763     Q_INVOKABLE void toggleConnectDriving(bool checked)
764     {
765         if (checked)
766         {
767             dataVec["Vehicle_Control_Unit.S1_ES_PGN61427.Control_Es_Relay
                    "] = 3;
768             IsConnectForDriving = true;
769             // stream << "Timestamp,CAN_ID\n";
770         }
771         else
772         {
773             IsConnectForDriving = false;
774             dataVec["Vehicle_Control_Unit.S1_ES_PGN61427.Control_Es_Relay
                    "] = 0;
775
776             // sevcon inverter control word set for all the 4 inverter
777             dataVec["Vehicle_Control_Unit.HC1_Demands.ControlWord"] = 6;
778             dataVec["Vehicle_Control_Unit.HC1_Demands_2.ControlWord"] =
                    6;
779             dataVec["Vehicle_Control_Unit.HC1_Demands_3.ControlWord"] =
                    6;
780             dataVec["Vehicle_Control_Unit.HC1_Demands_4.ControlWord"] =
                    6;
781         }
```

```
782     }
783     // possibly check if these keys exists before accesing to avoid
            crashes in the UI
784
785     Q_INVOKABLE void toggleACCharging(bool checked)
786     {
787         if (checked)
788         {
789             dataVec["Vehicle_Control_Unit.S1_ES_PGN61427.Control_Es_Relay
                    "] = 2;
790             IsConnectForDriving = true;
791             IsConnectedForACCharging = true;
792         }
793         else
794         {
795             dataVec["Vehicle_Control_Unit.S1_ES_PGN61427.Control_Es_Relay
                    "] = 0;
796             // dataVec[1].IsSent = false;
797             IsConnectForDriving = false;
798             IsConnectedForACCharging = false;
799         }
800     }
801     // possibly check if these keys exists before accesing to avoid
            crashes in the UI
802     Q_INVOKABLE void toggleDCCharging(bool checked)
803     {
804         if (checked)
805         {
806             dataVec["Vehicle_Control_Unit.S1_ES_PGN61427.Control_Es_Relay
                    "] = 5;
807             IsConnectForDriving = true;
808             IsConnectedForDCCharging = true;
809         }
810         else
811         {
812             dataVec["Vehicle_Control_Unit.S1_ES_PGN61427.Control_Es_Relay
                    "] = 0;
813             IsConnectForDriving = false;
814             IsConnectedForDCCharging = false;
```

```
815         }
816     }
817     // possibly check if these keys exists before accesing to avoid
            crashes in the UI
818     Q_INVOKABLE void toggleCrash(bool checked)
819     {
820         if (checked)
821         {
822             dataVec["Vehicle_Control_Unit.CN_PGN61483.Crash_Typ"] = 1;
823             IsConnectForDriving = true;
824         }
825         else
826         {
827             dataVec["Vehicle_Control_Unit.CN_PGN61483.Crash_Typ"] = 0;
828             IsConnectForDriving = false;
829         }
830     }
831 };
832 void splitMessagesByNodeName(
833     const std::unordered_map<std::string, Message> &inputMessages,
834     const std::vector<std::string> &group1NodeNames,
835     const std::vector<std::string> &group2NodeNames,
836     std::unordered_map<std::string, Message> &group1Messages,
837     std::unordered_map<std::string, Message> &group2Messages)
838 {
839     for (const auto &[key, message] : inputMessages)
840     {
841         const std::string &node = message.nodeName;
842 
843         if (std::find(group1NodeNames.begin(), group1NodeNames.end(),
                node) != group1NodeNames.end())
844         {
845             group1Messages[key] = message;
846         }
847         else if (std::find(group2NodeNames.begin(), group2NodeNames.end()
                , node) != group2NodeNames.end())
848         {
849             group2Messages[key] = message;
850         }
```

```
851         else
852         {
853             qDebug() << "Failed splitting message -- check here
                immediately";
854         }
855     }
856 }
857
858 int main(int argc, char *argv[])
859 {
860     set_qt_environment();
861     QGuiApplication app(argc, argv);
862
863     // Enable Qt Virtual Keyboard
864     qputenv("QT_IM_MODULE", QByteArray("qtvirtualkeyboard"));
865
866     QQmlApplicationEngine engine;
867
868     engine.rootContext()->setContextProperty("logManager", &logManager);
869
870     WifiManager wifiManager;
871     engine.rootContext()->setContextProperty("wifiManager", &wifiManager)
        ;
872     // Register SystemManager for QML
873     SystemManager systemManager;
874     engine.rootContext()->setContextProperty("systemManager", &
        systemManager);
875     logManager.addLog("Vehicle Initialised");
876
877     TreeModel treemodel;
878     engine.rootContext()->setContextProperty("treeModel", &treemodel);
879     // Create and expose TimeManager to QML
880     TimeManager timeManager;
881     engine.rootContext()->setContextProperty("timeManager", &timeManager)
        ;
882     SignalValueMap signalValueMapInstance;
883     // Expose to QML
884     engine.rootContext()->setContextProperty("signalValueMap", &
        signalValueMapInstance);
```

```
885     qmlRegisterType<DatabaseManager>("com.example.db", 1, 0, "
            DatabaseManager");
886     qmlRegisterType<MainController>("MainController", 1, 0, "
            MainController");
887     MainController mainController;
888     engine.rootContext()->setContextProperty("mainController", &
            mainController);
889
890     const QUrl url(mainQmlFile);
891     QObject::connect(
892         &engine, &QQmlApplicationEngine::objectCreated, &app,
893         [url](QObject *obj, const QUrl &objUrl)
894         {
895             if (!obj && url == objUrl)
896                 QCoreApplication::exit(-1);
897         },
898         Qt::QueuedConnection);
899
900     engine.addImportPath(QCoreApplication::applicationDirPath() + "/qml")
            ;
901     engine.addImportPath(":/");
902     engine.load(url);
903
904     if (engine.rootObjects().isEmpty())
905         return -1;
906
907     // Create table
908     DatabaseManager dbManager;
909
910     // // Read data into model
911     std::unordered_map<std::string, Message> messages;
912     QSqlQueryModel *model = dbManager.readData();
913     // Iterate through the rows in the model
914     for (int row = 0; row < model->rowCount(); ++row)
915     {
916         QString sheetName = model->index(row, model->record().indexOf("
                SheetName")).data().toString();
917         QString nodeName = model->index(row, model->record().indexOf("
                NodeName")).data().toString();
```

```
        if (sheetName != "Main" && sheetName != "nonshm")
        {
            continue; // Skip rows not belonging to the specified sheets
        }

        std::string pgn;
        long double value = 0.0;
        std::string signal;
        long double cycleTime = 0.0;
        long double deltaTime = 0.0;
        bool isSHM = false;
        std::string messageName;

        // Access columns based on their names or indexes
        QVariant pgnVar = model->index(row, model->record().indexOf("
            UniqueID")).data();
        QVariant valueVar = model->index(row, model->record().indexOf("
            Value")).data();
        QVariant signalVar = model->index(row, model->record().indexOf("
            Signal")).data();
        QVariant cycleTimeVar = model->index(row, model->record().indexOf
            ("CycleTime")).data();
        QVariant deltaTimeVar = model->index(row, model->record().indexOf
            ("DeltaTime")).data();
        QVariant isSHMVar = model->index(row, model->record().indexOf("
            IsSHM")).data();
        QVariant messageNameVar = model->index(row, model->record().
            indexOf("Message")).data();

        // Convert QVariant values into the appropriate types
        pgn = pgnVar.toString().toStdString();
        value = valueVar.toDouble(); // Convert QVariant to long double
        signal = signalVar.toString().toStdString();
        cycleTime = cycleTimeVar.toDouble();
        deltaTime = deltaTimeVar.toDouble();

        // Check for NaN in deltaTime and replace with 0.0 if necessary
        if (std::isnan(deltaTime))
        {
```

```
950             deltaTime = 0.0;
951         }
952 
953         isSHM = isSHMVar.toBool(); // Convert QVariant to bool
954         messageName = messageNameVar.toString().toStdString();
955 
956         // Store the data in the map
957         if (isSHM)
958         {
959             messages[pgn].SHM.emplace_back(value, cycleTime, deltaTime,
                    signal, messageName);
960         }
961         else
962         {
963             messages[pgn].SDM.emplace_back(value, cycleTime, deltaTime,
                    signal, messageName);
964         }
965         messages[pgn].sheetName = sheetName.toStdString();
966         messages[pgn].nodeName = nodeName.toStdString();
967     }
968     // Output results for debugging
969     for (const auto &pair : messages)
970     {
971         const auto &pgn = pair.first;
972         const auto &message = pair.second;
973 
974         qDebug() << "----------------------------------------";
975         qDebug() << "PGN:" << pgn;
976         qDebug() << "Sheet Name:" << QString::fromStdString(message.
                sheetName);
977         qDebug() << "Node Name:" << QString::fromStdString(message.
                nodeName);
978 
979         qDebug() << "\nSHM Entries:";
980         if (message.SHM.empty())
981         {
982             qDebug() << "  No SHM entries.";
983         }
984         else
```

```
985         {
986             for (const auto &shmEntry : message.SHM)
987             {
988                 qDebug().nospace()
989                     << "  Value: " << QString::number(shmEntry.value, 'f'
                            , 5)
990                     << ", Cycle Time: " << QString::number(shmEntry.
                            cycleTime, 'f', 5)
991                     << ", Delta Time: " << QString::number(shmEntry.
                            deltaTime, 'f', 5)
992                     << ", Signal: " << QString::fromStdString(shmEntry.
                            signal)
993                     << ", Message: " << QString::fromStdString(shmEntry.
                            message);
994             }
995         }
996 
997         qDebug() << "\nSDM Entries:";
998         if (message.SDM.empty())
999         {
1000            qDebug() << "  No SDM entries.";
1001        }
1002        else
1003        {
1004            for (const auto &sdmEntry : message.SDM)
1005            {
1006                qDebug().nospace()
1007                    << "  Value: " << QString::number(sdmEntry.value, 'f'
                            , 5)
1008                    << ", Cycle Time: " << QString::number(sdmEntry.
                            cycleTime, 'f', 5)
1009                    << ", Delta Time: " << QString::number(sdmEntry.
                            deltaTime, 'f', 5)
1010                    << ", Signal: " << QString::fromStdString(sdmEntry.
                            signal)
1011                    << ", Message: " << QString::fromStdString(sdmEntry.
                            message);
1012            }
1013        }
```

```
1014    }
1015
1016    qDebug() << "----------------------------------------";
1017
1018    // Parse DBC file
1019    QCanDbcFileParser fileParser;
1020    QString path = "/root/vehicle.dbc";
1021    QString str = QDir::toNativeSeparators(path); // Converts to correct
            format
1022
1023    const bool result = fileParser.parse(str);
1024    if (result)
1025    {
1026        logManager.addLog("DBC File Parsed Successfully:" + path);
1027    }
1028    else
1029    {
1030        logManager.addLog("Failed Parsing. Error:" + fileParser.
                errorString());
1031    }
1032    // CAN reciever logic sits here
1033    QJsonObject fsm = loadFSMFromFile("/root/fsm.json");
1034
1035    TimeDateResponder responder(signalValueMapInstance.m_signalValueMap,
            fsm);
1036    responder.initialize();
1037
1038    // fix to split the messages based on nodename column and spawn the
            new thread after splitting
1039    //  group 1:  vig, pdu, obc -- power data together as they are time
            specific
1040    //  group 2:  hvlp1, hvlp2, hvlp3, hvlp4, editron -- basically all
            inverter data together
1041    std::vector<std::string> group1 = {"vig", "pdu", "obc"};
1042    std::vector<std::string> group2 = {"hvlp1", "hvlp2", "hvlp3", "hvlp4"
            , "editron"};
1043
1044    std::unordered_map<std::string, Message> group1Messages;
1045    std::unordered_map<std::string, Message> group2Messages;
```

```
1046
1047     splitMessagesByNodeName(messages, group1, group2, group1Messages,
             group2Messages);
1048
1049     // Initialize the CanMessageSender thread
1050     CanMessageSenderGroup1 senderGroup1(fileParser, group1Messages); //
             main thread handling logic flow
1051     senderGroup1.start();                                             //
             This starts the thread
1052
1053     // below thread sends group2 message data -- if you change logic in
             the above thread make sure to revisit here
1054     CanMessageSenderGroup2 senderGroup2(fileParser, group2Messages); //
             main thread handling logic flow
1055     senderGroup2.start();                                             //
             This starts the thread
1056     return app.exec();
1057 }
1058
1059 #include "main.moc" // Add this if Q_OBJECT is used
```

Listing C.1: VCU Software Skeleton (main.cpp)

C.2 JSON-Based FSM Logic Configuration

The vehicle control logic is defined in a fully customizable and human-readable JSON file that specifies all valid states, events, and transitions for the finite state machine (FSM). This approach decouples control logic from C++ code, enabling developers or engineers to easily define and update startup, shutdown, charging, and fault-handling sequences without modifying source code.

A simplified version of the FSM configuration is shown below:

```
1 {
2     "initialState": "On Boot",
3     "states": [
4       {
5         "name": "VCU Initilization",
6         "transitions": [
```

```
7           {
8             "condition": {
9               "Vehicle_Control_Unit.HVESSC1_PGN6912_Rx1.HS_OperConsent_Rx
                    1": 1
10            },
11            "actions": [
12            ],
13            "nextState": "Start"
14          }
15        ]
16      },
17      {
18        "name": "Start",
19        "transitions": [
20          {
21            "condition": {
22              "all": [
23                { "key": "HVLP_SA71.HS2_Status_Feedback.InverterStatus",
                      "eq": 1 },
24                { "key": "HVLP_SA71.HS2_Status_Feedback.InverterStatus",
                      "eq": 13 },
25                { "key": "HVLP_SA72.HS2_Status_Feedback_2.InverterStatus"
                      , "eq": 1 },
26                { "key": "HVLP_SA72.HS2_Status_Feedback_2.InverterStatus"
                      , "eq": 13 },
27                { "key": "HVLP_SA73.HS2_Status_Feedback_3.InverterStatus"
                      , "eq": 1 },
28                { "key": "HVLP_SA73.HS2_Status_Feedback_3.InverterStatus"
                      , "eq": 13 },
29                { "key": "HVLP_SA74.HS2_Status_Feedback_4.InverterStatus"
                      , "eq": 1 },
30                { "key": "HVLP_SA74.HS2_Status_Feedback_4.InverterStatus"
                      , "eq": 13 },
31                { "key": "VIC.HVESSS1_PGN61590.HS_HiUBusCnctnSts", "eq":
                      14},
32                { "key": "PDU.PDU_Tx_1.byPDU_Status", "ne": 1 },
33                { "key":"Editron.ST_MOT_2.st_mot_fault", "eq": 2},
34                { "key": "OBC.OBC.OBC_Charger_Ready", "eq": 1}
35              ]
```

```
      },
      "actions": [
        { }
      ],
      "nextState": "ShutDown"
    },
    {
      "condition": {
        "VIC.HVESSS1_PGN61590.HS_HiUBusCnctnSts": 14
      },
      "actions": [
        { }
      ],
      "nextState": "Battery Fault"
    },
    {
      "condition": {
        "any": [
          { "key": "PDU.PDU_Tx_1.byPDU_Status", "ne": 1 }
        ]
      }
      ,
      "actions": [
        { }
      ],
      "nextState": "PDU Fault"
    },
    {
      "condition": {
        "any": [
          { "key": "HVLP_SA71.HS2_Status_Feedback.InverterStatus",
              "eq": 1 },
          { "key": "HVLP_SA71.HS2_Status_Feedback.InverterStatus",
              "eq": 13 },
          { "key": "HVLP_SA72.HS2_Status_Feedback_2.InverterStatus"
              , "eq": 1 },
          { "key": "HVLP_SA72.HS2_Status_Feedback_2.InverterStatus"
              , "eq": 13 },
```

```
                { "key": "HVLP_SA73.HS2_Status_Feedback_3.InverterStatus"
                    , "eq": 1 },
                { "key": "HVLP_SA73.HS2_Status_Feedback_3.InverterStatus"
                    , "eq": 13 },
                { "key": "HVLP_SA74.HS2_Status_Feedback_4.InverterStatus"
                    , "eq": 1 },
                { "key": "HVLP_SA74.HS2_Status_Feedback_4.InverterStatus"
                    , "eq": 13 }
              ]
            },
            "actions": [
              { }
            ],
            "nextState": "HVLP Inverter Fault"
          },
          {
            "condition": {
              "Editron.ST_MOT_2.st_mot_fault": 2
            },
            "actions": [
              {  }
            ],
            "nextState": "Editron Inverter Fault"
          },
          {
            "condition": {
              "OBC.OBC.OBC_Charger_Ready": 1
            },
            "actions": [
              {  }
            ],
            "nextState": "OBC Fault"
          },
          {
            "condition": {
              "VIC.HVESSS1_PGN61590.HS_HiUBusCnctnSts": 1
            },
            "actions": [
```

```
104             { "signal": "Vehilce_Control_Unit.PDU_Rx_1.
                    Close_GroupA_Contactors", "value": 1 },
105             { "signal": "Vehicle_Control_Unit.PDU_Rx_1.
                    Close_GroupB_Contactors", "value": 1 },
106             { "signal": "Vehicle_Control_Unit.HC1_Demands.ControlWord",
                     "value": 6 },
107             { "signal": "Vehicle_Control_Unit.HC1_Demands_2.ControlWord
                    ", "value": 6 },
108             { "signal": "Vehicle_Control_Unit.HC1_Demands_3.ControlWord
                    ", "value": 6 },
109             { "signal": "Vehicle_Control_Unit.HC1_Demands_4.ControlWord
                    ", "value": 6 },
110             { "signal": "Vehicle_Control_Unit.CMD_MOT_0.cmd_mot_request
                    ", "value": 1 },
111             { "signal": "Vehicle_Control_Unit.BMS.BMS_Control", "value"
                    : 0 },
112             { "signal": "Vehicle_Control_Unit.BMS.BMS_Max_Voltage", "
                    value": 700 },
113             { "signal": "Vehicle_Control_Unit.BMS.BMS_Max_Current", "
                    value": 60 },
114             { "signal": "Vehicle_Control_Unit.VCU.VCU_DCDC_Command", "
                    value": 1 },
115             { "signal": "Vehicle_Control_Unit.VCU.VCU_DCDC_MaxI", "
                    value": 3 },
116             { "signal": "Vehicle_Control_Unit.VCU.VCU_DCDC_SetV", "
                    value": 12 }
117           ],
118           "nextState": "Connect for Driving"
119         },
120         {
121           "condition": {
122             "any": [
123               { "key": "Vehicle_Control_Unit.HVESSC1_PGN6912_Rx1.
                      HS_IntChrgrSts", "eq": 1 },
124               { "key": "VIC.HVESSS1_PGN61590.HS_HiUBusCnctnSts", "gt":
                      4 }
125             ]
126           },
127           "actions": [
```

```
128             { "signal": "Vehilce_Control_Unit.PDU_Rx_1.
                    Close_GroupA_Contactors", "value": 0 },
129             { "signal": "Vehicle_Control_Unit.PDU_Rx_1.
                    Close_GroupB_Contactors", "value": 1 },
130             { "signal": "Vehicle_Control_Unit.BMS.BMS_Control", "value"
                    : 0 },
131             { "signal": "Vehicle_Control_Unit.BMS.BMS_Max_Voltage", "
                    value": 700 },
132             { "signal": "Vehicle_Control_Unit.BMS.BMS_Max_Current", "
                    value": 60 },
133             { "signal": "Vehicle_Control_Unit.VCU.VCU_DCDC_Command", "
                    value": 1 },
134             { "signal": "Vehicle_Control_Unit.VCU.VCU_DCDC_MaxI", "
                    value": 3 },
135             { "signal": "Vehicle_Control_Unit.VCU.VCU_DCDC_SetV", "
                    value": 12 }
136           ],
137           "nextState": "Connect for Charging"
138         },
139         {
140           "condition": {
141           },
142           "actions": [
143             { "signal": "Vehilce_Control_Unit.PDU_Rx_1.
                    Close_GroupA_Contactors", "value": 0 },
144             { "signal": "Vehicle_Control_Unit.PDU_Rx_1.
                    Close_GroupB_Contactors", "value": 0 },
145             { "signal": "Vehicle_Control_Unit.BMS.BMS_Control", "value"
                    : 1 },
146             { "signal": "Vehicle_Control_Unit.VCU.VCU_DCDC_Command", "
                    value": 0 },
147             { "signal": "Vehicle_Control_Unit.HC1_Demands.ControlWord",
                     "value": 6 },
148             { "signal": "Vehicle_Control_Unit.HC1_Demands_2.ControlWord
                    ", "value": 6 },
149             { "signal": "Vehicle_Control_Unit.HC1_Demands_3.ControlWord
                    ", "value": 6 },
150             { "signal": "Vehicle_Control_Unit.HC1_Demands_4.ControlWord
                    ", "value": 6 },
```

```
151                 { "signal": "Vehicle_Control_Unit.CMD_MOT_0.cmd_mot_request
                        ", "value": 0 }
152               ],
153               "nextState": "Stop"
154             }
155           ]
156         },
157         {
158           "name": "Stop",
159           "transitions": [
160             {
161               "condition": {
162                 "all": [
163                   { "key": "Vehicle_Control_Unit.HVESSC1_PGN6912_Rx1.
                          HS_OperConsent_Rx1", "eq": 1 }
164                 ]
165               },
166               "actions": [
167                 { }
168               ],
169               "nextState": "Start"
170             }
171           ]
172         },
173         {
174           "name": "Connect for Driving",
175           "transitions": [
176             {
177               "condition": {
178                 "all": [
179                   { "key": "Vehicle_Control_Unit.HC1_Demands_2.ControlWord"
                          , "eq": 6 },
180                   { "key": "Vehicle_Control_Unit.HC1_Demands_2.ControlWord"
                          , "eq": 6 },
181                   { "key": "Vehicle_Control_Unit.HC1_Demands_2.ControlWord"
                          , "eq": 6 },
182                   { "key": "Vehicle_Control_Unit.HC1_Demands_2.ControlWord"
                          , "eq": 6 }
183                 ]
```

```
184             },
185             "actions": [
186               { "signal": "Vehicle_Control_Unit.HC1_Demands.ControlWord",
                      "value": 3 },
187               { "signal": "Vehicle_Control_Unit.HC1_Demands_2.ControlWord
                      ", "value": 3 },
188               { "signal": "Vehicle_Control_Unit.HC1_Demands_3.ControlWord
                      ", "value": 3 },
189               { "signal": "Vehicle_Control_Unit.HC1_Demands_4.ControlWord
                      ", "value": 3 }
190             ],
191             "nextState": "Connect for Driving"
192           },
193           {
194             "condition": {
195               "all": [
196                 { "key": "Vehicle_Control_Unit.HC1_Demands_2.ControlWord"
                        , "eq": 3 },
197                 { "key": "Vehicle_Control_Unit.HC1_Demands_2.ControlWord"
                        , "eq": 3 },
198                 { "key": "Vehicle_Control_Unit.HC1_Demands_2.ControlWord"
                        , "eq": 3 },
199                 { "key": "Vehicle_Control_Unit.HC1_Demands_2.ControlWord"
                        , "eq": 3 }
200               ]
201             },
202             "actions": [
203               { "signal": "Vehicle_Control_Unit.HC1_Demands.ControlWord",
                      "value": 5 },
204               { "signal": "Vehicle_Control_Unit.HC1_Demands_2.ControlWord
                      ", "value": 5 },
205               { "signal": "Vehicle_Control_Unit.HC1_Demands_3.ControlWord
                      ", "value": 5 },
206               { "signal": "Vehicle_Control_Unit.HC1_Demands_4.ControlWord
                      ", "value": 5 }
207             ],
208             "nextState": "Connect for Driving"
209           },
210           {
```

```
            "condition": {
              "all": [
                { "key": "VIC.HVESSD2_PGN61585.HS_FastUpdSOC", "gt": 89 }
                    ,
                { "key": "Vehicle_Control_Unit.BMS.BMS_Control", "eq": 0
                    }
              ]
            },
            "actions": [
              { "signal": "Vehicle_Control_Unit.BMS.BMS_Control", "value"
                  : 1 }
            ],
            "nextState": "Connect for Driving"
          },
          {
            "condition": {
              "VIC.HVESSS1_PGN61590.HS_HiUBusCnctnSts": 14
            },
            "actions": [
              { }
            ],
            "nextState": "Battery Fault"
          },
          {
            "condition": {
              "any": [
                { "key": "PDU.PDU_Tx_1.byPDU_Status", "ne": 1 }
              ]
            }
            ,
            "actions": [
              { }
            ],
            "nextState": "PDU Fault"
          },
          {
            "condition": {
              "any": [
```

```
246                 { "key": "HVLP_SA71.HS2_Status_Feedback.InverterStatus",
                        "eq": 1 },
247                 { "key": "HVLP_SA71.HS2_Status_Feedback.InverterStatus",
                        "eq": 13 },
248                 { "key": "HVLP_SA72.HS2_Status_Feedback_2.InverterStatus"
                        , "eq": 1 },
249                 { "key": "HVLP_SA72.HS2_Status_Feedback_2.InverterStatus"
                        , "eq": 13 },
250                 { "key": "HVLP_SA73.HS2_Status_Feedback_3.InverterStatus"
                        , "eq": 1 },
251                 { "key": "HVLP_SA73.HS2_Status_Feedback_3.InverterStatus"
                        , "eq": 13 },
252                 { "key": "HVLP_SA74.HS2_Status_Feedback_4.InverterStatus"
                        , "eq": 1 },
253                 { "key": "HVLP_SA74.HS2_Status_Feedback_4.InverterStatus"
                        , "eq": 13 }
254               ]
255             },
256             "actions": [
257               { }
258             ],
259             "nextState": "HVLP Inverter Fault"
260           },
261           {
262             "condition": {
263               "Editron.ST_MOT_2.st_mot_fault": 2
264             },
265             "actions": [
266               {  }
267             ],
268             "nextState": "Editron Inverter Fault"
269           },
270           {
271             "condition": {
272               "OBC.OBC.OBC_Charger_Ready": 1
273             },
274             "actions": [
275               {  }
276             ],
```

```
277             "nextState": "OBC Fault"
278           }
279         ]
280       },
281       {
282         "name": "Connect for Charging",
283         "transitions": [
284           {
285             "condition": {
286               "all": [
287                 { "key": "VIC.HVESSD2_PGN61585.HS_FastUpdSOC", "gt": 89 }
                      ,
288                 { "key": "Vehicle_Control_Unit.BMS.BMS_Control", "eq": 0
                      }
289               ]
290             },
291             "actions": [
292               { "signal": "Vehicle_Control_Unit.BMS.BMS_Control", "value"
                    : 1 }
293             ],
294             "nextState": "Connect for Charging"
295           },
296           {
297             "condition": {
298               "VIC.HVESSS1_PGN61590.HS_HiUBusCnctnSts": 14
299             },
300             "actions": [
301               { }
302             ],
303             "nextState": "Battery Fault"
304           },
305           {
306             "condition": {
307               "any": [
308                 { "key": "PDU.PDU_Tx_1.byPDU_Status", "ne": 1 }
309               ]
310             }
311             ,
312             "actions": [
```

```
313                 { }
314               ],
315               "nextState": "PDU Fault"
316             },
317             {
318               "condition": {
319                 "OBC.OBC.OBC_Charger_Ready": 1
320               },
321               "actions": [
322                 {  }
323               ],
324               "nextState": "OBC Fault"
325             }
326           ]
327         },
328         {
329           "name": "HVLP Inverter Fault",
330           "transitions": [
331             {
332               "condition": {
333                 "all": [
334                   { "key": "HVLP_SA71.HS2_Status_Feedback.InverterStatus",
                        "eq": 8 },
335                   { "key": "HVLP_SA72.HS2_Status_Feedback_2.InverterStatus"
                        , "eq": 8 },
336                   { "key": "HVLP_SA73.HS2_Status_Feedback_3.InverterStatus"
                        , "eq": 8 },
337                   { "key": "HVLP_SA74.HS2_Status_Feedback_4.InverterStatus"
                        , "eq": 8 }
338                 ]
339               },
340               "actions": [
341                 {  }
342               ],
343               "nextState": "Start"
344             }
345           ]
346         },
347         {
```

```
348         "name": "Editron Inverter Fault",
349         "transitions": [
350           {
351             "condition": {
352               "Editron.ST_MOT_2.st_mot_fault": 1
353             },
354             "actions": [
355               {  }
356             ],
357             "nextState": "Start"
358           }
359         ]
360       },
361       {
362         "name": "Battery Fault",
363         "transitions": [
364           {
365             "condition": {
366               "all": [
367                 { "key": "VIC.HVESSS1_PGN61590.HS_HiUBusCnctnSts", "ne":
                      14 }
368               ]
369             },
370             "actions": [
371               {  }
372             ],
373             "nextState": "Start"
374           }
375         ]
376       },
377       {
378         "name": "OBC Fault",
379         "transitions": [
380           {
381             "condition": {
382               "all": [
383                 { "key": "OBC.OBC.OBC_Charger_Ready", "eq": 0 }
384               ]
385             },
```

```
            "actions": [
              { }
            ],
            "nextState": "Start"
          }
        ]
      },
      {
        "name": "PDU Fault",
        "transitions": [
          {
            "condition": {
              "any": [
                { "key": "PDU.PDU_Tx_1.byPDU_Status", "eq": 1 }
              ]
            },
            "actions": [
              {  }
            ],
            "nextState": "Start"
          }
        ]
      },
      {
        "name": "ShutDown",
        "transitions": []
      }
    ]
  }
```

Listing C.2: FSM JSON Configuration

Each transition includes: - ‘from‘: current state - ‘to‘: target state - ‘on‘: event name (trigger) - ‘condition‘: signal-based condition using runtime ‘dataVector‘ values

To customize the FSM: - Update or add ‘states‘, ‘events‘, and ‘transitions‘ - Use keys in the format ‘Node.Message.Signal‘ to refer to specific CAN signals - Supported conditions: ‘eq‘, ‘gt‘, ‘lt‘, ‘neq‘, etc.

This file is loaded at runtime and parsed using Qt’s ‘QJsonObject‘, with state transitions evaluated periodically inside a dedicated FSM thread.

Figure C.1 illustrates the finite state machine (FSM) governing the high-level power-up and fault-handling logic of the vehicle control system. The FSM begins in the *On Boot* state and transitions through key operational phases such as *VCU Initialization*, *Start*, *Connect for Charging*, and *Connect for Driving*, depending on the evaluation of incoming signal conditions. Each transition is triggered based on real-time CAN signals such as inverter statuses, charger readiness, and contactor states. Fault-specific branches—like *HVLP Inverter Fault*, *Editron Inverter Fault*, *Battery Fault*, and *OBC Fault*—enable the system to gracefully handle anomalies and recover to the *Start* state once the fault conditions are resolved. This FSM provides a deterministic framework for initializing, driving, charging, and safely shutting down the vehicle.

Figure C.1: Finite State Machine

C.3 QML Dashboard Entry Point

The user interface for real-time monitoring and control is developed using QML and launched from a single entry file: ‘main.qml‘. The dashboard is modular and includes five primary tabs: 1. Vehicle Control 2. Signal Dashboard 3. ECU Setup 4. CAN Data 5. Debugging Tools

Each tab is defined as a standalone QML component and loaded dynamically into

the ‘TabView‘ structure, allowing easy customization and maintenance.

```
import QtQuick
import QtQuick.Layouts
import QtQuick.Controls
import Qt.labs.qmlmodels

Item {
    width: 720
    height: 600

    // Background
    Image {
        id: background
        anchors.fill: parent
        source: "images/back_195_184.png"
    }

    // CST Date and Time Display Container
    Rectangle {
        id: dateTimeContainer
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.top: parent.top
        anchors.topMargin: 5
        anchors.horizontalCenterOffset: 1
        radius: 10
        color: "#333333" // Dark background
        width: 718
        height: 50
        border.color: "#ffffff" // White border
        border.width: 1

        // Shadow effect (using multiple rectangles to simulate shadow)
        Rectangle {
            anchors.fill: parent
            color: "black"
            radius: 10
            opacity: 0.3
            width: parent.width
            height: parent.height
```

```
39              anchors.topMargin: 5
40              anchors.leftMargin: 5
41          }
42
43          // Date and Time Text
44          Text {
45              id: dateTimeDisplay
46              anchors.centerIn: parent
47              font.pixelSize: 18
48              font.bold: true
49              color: "#ffffff" // White text
50              text: timeManager.currentTime // Bind to the C++ TimeManager
51              horizontalAlignment: Text.AlignHCenter
52              verticalAlignment: Text.AlignVCenter
53              anchors.verticalCenterOffset: 0
54              anchors.horizontalCenterOffset: -223
55          }
56      }
57      // Battery Icon and Percentage Display
58      BatteryDisplay {
59          id: batteryDisplay
60      }
61      // Menu Layout
62      RowLayout {
63          id: top_menu
64          anchors.horizontalCenter: parent.horizontalCenter
65          anchors.top: dateTimeContainer.bottom
66          anchors.topMargin: -8
67          spacing: 8.5
68
69          // Your custom buttons
70          Vehicle {
71              id: customButton0
72              width: 120
73              checked: true
74              autoExclusive: true
75              onClicked: stackLayout.currentIndex = 0
76          }
77
```

```
78          Dashboard {
79              id: customButton1
80              width: 120
81              autoExclusive: true
82              onClicked: stackLayout.currentIndex = 1
83          }
84
85          ECUSetup {
86              id: customButton2
87              width: 120
88              autoExclusive: true
89              onClicked: stackLayout.currentIndex = 2
90          }
91
92          CANData {
93              id: customButton3
94              width: 120
95              autoExclusive: true
96              onClicked: stackLayout.currentIndex = 3
97          }
98
99          Debug {
100             id: customButton4
101             width: 120
102             autoExclusive: true
103             onClicked: stackLayout.currentIndex = 4
104         }
105     }
106
107     StackLayout {
108         id: stackLayout
109         x: 0
110         y: 129
111         width: 720
112         height: 509
113
114         VehicleStack {
115             id: effectStack1
116         }
```

```
117
118         DashStack {
119             id: effectStack2
120         }
121
122         ECUStack {
123             id: effectStack3
124         }
125
126         CANStack {
127             id: effectStack4
128         }
129
130         DebugStack {
131             id: effectStack5
132         }
133     }
134 }
```

Listing C.3: Dashboard Entry Point (main.qml)

Each tab file (‘*.qml‘) is self-contained and interacts with the backend through Qt’s signal-slot mechanism and ‘QVariantMap‘-based signal data. Developers can add or replace tabs without modifying the main file, making this system highly modular.