TECHNISCHE UNIVERSITÄT
CHEMNITZ

Fakultät für Informatik

CSR-25-01

AUTOSAR Software Component for Atomic Straight Driving Patterns

Md. Ali Awlad · Hasan Saadi Jaber Aljzaere · Wolfram Hardt

März 2025

Chemnitzer Informatik-Berichte

AUTOSAR Software Component for Atomic Straight Driving Patterns

Master Thesis

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

Dept. of Computer Science
Chair of Computer Engineering

Submitted by: Md. Ali Awlad
Student ID: 635459
Date: 05.12.2024

Supervising tutor: Prof. Dr. Dr. h. c. Wolfram Hardt
Hasan Saadi Jaber Aljzaere

Abstract

The main focus of this thesis is the development of an AUTOSAR Software Component (SWC) to operate an Advanced Driver Assistance System (ADAS) demonstrator's engine for executing atomic straight driving patterns. The AUTOSAR classic architecture is the most commonly used in the automotive domain and consists of three basic layers: Basic Software (BSW), Application, and Runtime Environment (RTE). The straight driving pattern software component is developed to ensure adaptability within this architecture. The software component is controlled at different distances and speeds, and it receives and processes CAN bus messages to control the demonstrator’s actuators.

The computer engineering professorship of Chemnitz University of Technology (TU Chemnitz) offers extensive opportunities for working with AUTOSAR within its Automotive Software Engineering (ASE) Lab, which has different ADAS demonstrators. TUCminiCar is the latest ADAS demonstrator within these, developed under the AUTOSAR 4.0 version. This ADAS demonstrator has used the Infineon AURIX TC387 microcontroller, which supports AUTOSAR and CAN communication. The thesis scope is mainly divided into two parts: analyzing AUTOSAR modules within this demonstrator and designing and integrating the required software component to execute the atomic straight driving pattern functionality. The thesis goal completion depends on the successful operation of the atomic straight driving capability of the TUCminiCar within the AUTOSAR development framework with higher accuracy.

Since the atomic straight driving pattern is the most fundamental part of ADAS, the thesis outcome can be used as a foundation to implement additional autonomous driving patterns within the demonstrator by simply updating or adjusting the software component.

Keywords: Automotive Open System Architecture (AUTOSAR), Advanced Driver Assisting System (ADAS), Atomic Straight Driving Pattern, Electronic Control Unit (ECU), Software Component (SWC), Controller Area Network (CAN).

Acknowledgement

I would like to take this opportunity to thank all those who supported and guided me throughout the process of completing this thesis.

I would like to express my most overwhelming gratitude to my first supervisor, the honorable head of the department, “Professor Dr. Dr. h. c. Wolfram Hardt”, for his enormous guidance, valuable ideas, precious time, and effort.

Secondly, I would like to extend my equal appreciation to my second supervisor, “Mr. Hasan Saadi Jaber Aljzaere,” for his helpful suggestions and support in receiving important ideas during the whole thesis work. His guidance, from internship to thesis work, helped me a lot in laying the foundation for working with AUTOSAR.

Lastly, I would like to thank “Mr. Murat Sevil,” who has been working as a HiWi in the ASE lab. His technical advice and practical assistance during implementation and testing were most valuable.

I want to thank all of my department teachers, family, and friends for their continuous support, understanding, and motivation. The list is too long to include here. Thanks for all being part of this journey.

Content

List of Figures

List of Tables

List of Abbreviations

AUTOSAR	Automotive Open System Architecture
ADAS	Advanced Driver Assisting System
SWC	Software Component
BSW	Basic Software
CAN	Controller Area Network
RTE	Runtime Environment
ECU	Electronic Control Unit
PWM	Pulse-Width Modulation
Sonar	Sound Navigation and Ranging
SIL	Software-In-Loop
DAP	Debug Access Port
DLC	Data Length Code
VFB	Virtual Functional Bus
MCAL	Microcontroller Abstraction Layer
ICU	Input Capture Unit
ADC	Analogue Digital Converter
DIO	Digital Input Output
GPIO	General-Purpose Input Output
API	Application Programming Interface
OBD	On-board diagnostics
PDU	Protocol Data Unit
PduR	Protocol Data Unit Router
CanIf	CAN Interface
RPort	Required Port
PPort	Provided Port
V-ECU	Virtual Electronic Control Unit
RC	Radio-Controlled
E2E	End-to-End
IDE	Identifier Extension
SRS	Substitute Remote Request
CDD	Complex Device Driver
DC	Direct Current

1 Introduction

Autonomous vehicles are showing rapid growth in the global market. An international standard must be maintained to maintain the functionality of different ECUs and related software within different vehicles and ensure standardized implementation across different regions. Automotive Open System Architecture (AUTOSAR) classic is the most widely used architecture in the automotive domain to develop and maintain various types of vehicle software. The classic AUTOSAR architecture has three main layers: application, runtime environment (RTE), and basic software (BSW).[1] The application layer is hardware-independent, where the application software components are placed to control different applications. The RTE layer interfaces this hardware-independent application layer and hardware-dependent basic software (BSW) layers.[1]

AUTOSAR was established in 2003 to save production and development costs and create reusable software for different ECUs.[2] Based on this, the TU Chemnitz lab also facilitates the latest ADAS demonstrator, “TUCminiCar,” which can be configured and tested for different autonomous driving scenarios. The TUCminiCar is designed and constructed based on an Infineon evaluation board KIT_A2G_TC387_3V3_TFT, which has a tri-core microcontroller (AURIX TC387) that supports AUTOSAR. The eval board also facilitates CAN or Ethernet communication to send or receive data from a tester. The ASE lab of TU Chemnitz also provides simulation tools like VEOS to test the functionality of created software components within virtual ECUs. It is necessary to develop and test the target software component (SWC) through a simulation process before integrating it into the "TUCminiCar" to test the functionality of atomic straight driving patterns. The concept of atomic driving is to divide a straight path into the smallest sections and operate the car within these sections without any interruption.[3] Throughout this thesis, firstly, AUTOSAR classic architecture will be analyzed, a software component for atomic straight driving will be created, and finally, one straight path will be broken up into the five smallest segments and tested autonomous functionality within these segments, including the whole path segment. In addition, one safety critical criterion, obstacle detection and action after detection, will also be tested.

This introduction chapter will provide an in-depth overview of the following sections: the motivation to work on the thesis topic (sub-section 1.1) and the Problem Statement (sub-section 1.2).

1.1 Motivation

The number of road accidents is increasing day by day around the world. The report in 2023, as presented by the World Health Organization (WHO), stated that more than 1 million people died from road accidents throughout different regions. It has been declared that road safety is a critical global issue, still two people die every 1 minute. Traffic accidents continue to be the biggest cause of death for young people aged under 30 years. [4] Human error while driving any vehicle remains one of the main reasons for these accidents. Technical faults like brake failure and other vehicle parts errors are also responsible for accidents, but those are much less so than human error. [5] The GIDAS accident database in the figure below claims that "Human error" accounts for 93.5% of traffic accidents.[6]

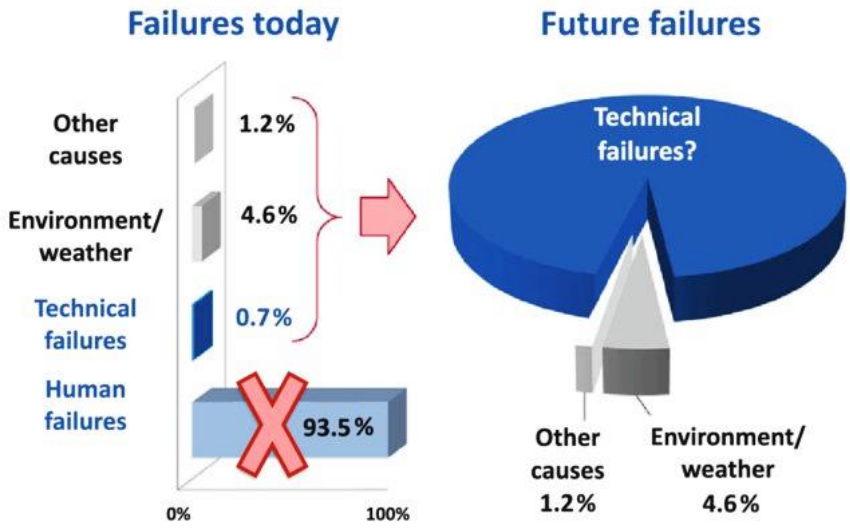

Figure 1.1: Accident Status in Percentage due to Human Error.[6]

With the transition from human-based driving to autonomous driving, this large segment of errors can be mitigated, and thus, future failures due to humans can be significantly reduced. The top three human errors that result in vehicle accidents are Inattentiveness, Speeding, and Improper lookout.[7] These life-endangered errors can be reduced by increasing self-driving cars, which will improve ADAS implementation.

With the advancement of the Advanced Driver Assisting System (ADAS), the automotive industry has experienced large market share growth, which will be significant soon. According to the “Fortune Business Insights” forecast, the autonomous vehicle market will hit around 13,632.4 billion USD by the year 2030, which is a 32.3% growth rate during the forecast period from 2022 to 2030. [8]

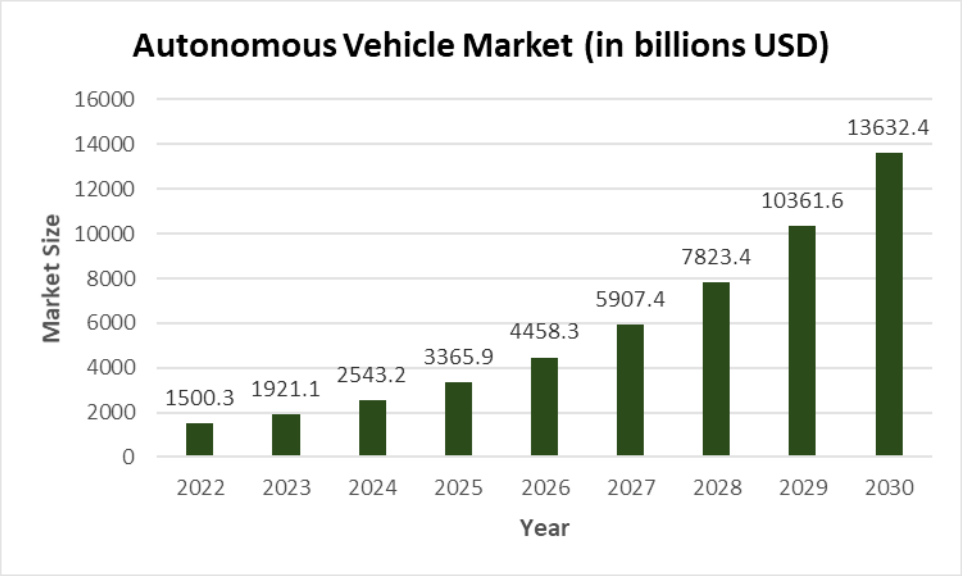

Figure 1.2: Growth of the Global Autonomous Vehicle Market.[8]

Considering the importance of autonomous vehicles in the automotive domain, the Computer Engineering professorship of TU Chemnitz has been working with ADAS demonstrators for a long time to help its students, especially those in Automotive Software Engineering, get used to the real-time environment of the automotive domain. Yellow Car (2010), Black Pearl (2018), and TUCminiCar (2024) are the demonstrators that operate within AUTOSAR architecture.

The atomic straight driving patterns will be implemented and tested on the “TUCminiCar” demonstrator in this thesis. Atomic straight driving is the most fundamental step, providing the foundation for going towards autonomous vehicles' different complex driving patterns. This core driving capacity ensures autonomous functionality within a car, which allows vehicle control mechanisms to be tested at different speeds and distances. Controller Area Network (CAN) communication is the most reliable method for sending or receiving parameters like speed or distance in the automotive domain, which is used to send/receive data from or to the ECU and tester. With CAN message format, atomic driving parameters, like speed and distance, can be sent from the tester to the ADAS demonstrator “TUCminiCar” to actuate the actuators of the car.

Maintaining the AUTOSAR development process framework for the desired software component will ensure reusability and adaptability for future related development. The following sub-section will describe the thesis's problem statement in detail.

1.2 Problem Statement

The thesis aims to create a software component that performs atomic straight drive functionality for an ADAS demonstrator within the AUTOSAR framework. The main challenges would be ensuring real-time performance and higher accuracy since the functionality will be tested within a toy car. Maintaining the standard implementation procedure within a miniature created based on a real-world environment where the real-world vehicle components will not be present is always difficult. While AUTOSAR standards are traditionally applied to full-scale automotive applications, transforming these specifications into mini cars demands a detailed problem analysis that accounts for more control, precision, and adaptability.

During the implementation of ADAS functionality in a miniaturized environment, it's crucial to ensure that the hardware components, like sensors for detecting obstacles (such as ultrasonic sensors) and actuators for controlling speed and steering (DC and Servo Motor), are precisely integrated. The software component needs to be highly responsive and adapt in real-time. Otherwise, even minor variations can significantly impact the vehicle's ability to drive straight due to the reduced scale. In general, this thesis aims to deal with the following particular problems:

Integration and Validation: Creating a mechanism by adapting AUTOSAR software structures for evaluating the component's straight-driving functionality in a test environment replicating a real-world environment.

Precision Control: Ensuring that the software component can accurately process actuator and sensor data to make real-time adjustments that maintain straight driving despite the limitations of mini-car hardware.

Higher Accuracy: Achieving higher accuracy for the actuator and sensor data processing to maintain accurate straight-line driving.

The thesis combines the identified challenges by designing, implementing, and validating an AUTOSAR software component supporting atomic straight-driving functions. This includes integrating sensor and actuator data for real-time control and applying mechanisms to maintain vehicle control. The next chapter will describe the Technical Background of the thesis topic.

2 Technical Background

This section will describe the foundation technologies for developing, integrating, and testing the AUTOSAR software component. The following sub-sections will briefly cover the AUTOSAR Standard Overview (2.1), ADAS Demonstrator under AUTOSAR (2.2), Atomic Straight Driving Core Concept (2.3), and Development and Testing Environment (2.4).

2.1 AUTOSAR Standard Overview

AUTomotiveOpen System Architecture (AUTOSAR) is the globally standardized automotive software architecture for manufacturers, suppliers, and developers. [9] Several manufacturers developed this architecture to produce a uniform software architecture that works independently with different electronic control units (ECUs). The AUTOSAR standard avoids re-creating software for identical purposes by separating hardware-specific layers. Depending on this idea, different companies can develop related software for different ECUs, and these can be run together in a car; thus, it is called hardware-independent architecture.

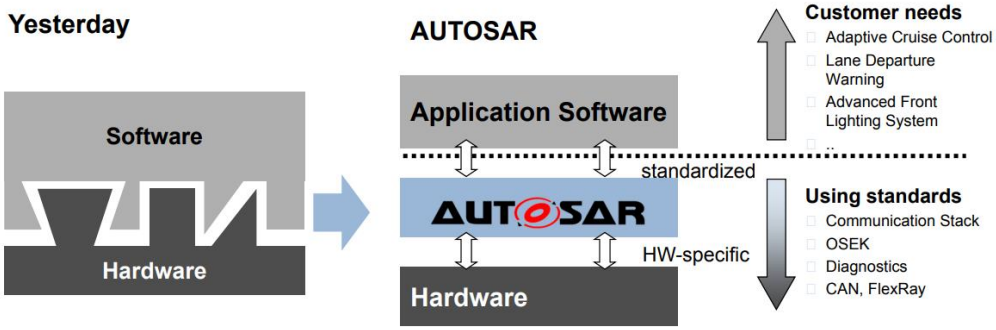

Figure 2.1: AUTOSAR Hardware-Independent Architecture.[10]

There are two basic AUTOSAR standards in terms of software architecture: one is classic, and the other one is adaptive. The classic AUTOSAR is used for embedded systems with hard real-time, whereas ECUs are used for the core components of a vehicle. The AUTOSAR classic architecture is mainly built on three software layers: Application, Runtime Environment (RTE), and Basic Software (BSW).[10] The main concept of the architecture is to separate the hardware-independent application software layer from the hardware-oriented basic software (BSW) layer with the help of the Runtime Environment (RTE) layer. [1] Due to the separation between different components as a layered structure, it is called a layered architecture. Different application software components (SWCs) used for different control mechanisms are created in the application layer of the architecture. These application software

components (SWCs) connect with different components placed in the basic software layer (BSW), which is hardware-dependent through the RTE layer. The BSW layer then communicates with different hardware devices using different communication protocols, e.g., CAN, to send or receive instructions.

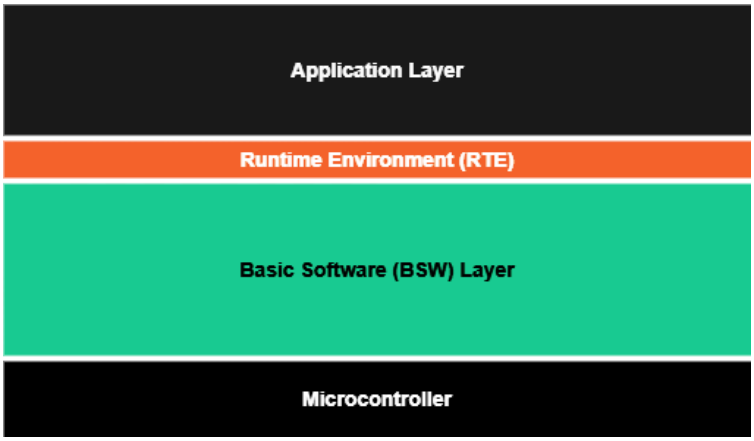

Figure 2.2: AUTOSAR Layered Architecture. [10]

The software component that needs to be created to control the car engine for atomic straight drive must be placed in the application layer of the system architecture.

2.2 ADAS Demonstrator under AUTOSAR

Advanced Driver Assistance Systems (ADAS) have had a huge impact on the advancement of modern vehicles. ADAS facilitates many safety features in real-world autonomous vehicles, like collision avoidance, adaptive cruise control, lane change assistance, blind spot detection, and driverless driving functionality. Chemnitz University of Technology's computer engineering professorship has been working for a long time to experience the real-time environment opportunity for its students to work with different ADAS demonstrators. Apart from different AUTOSAR-based simulation tools, the department has a well-developed infrastructure to test and validate demonstration cars. The latest inclusion in the department is "TUCminiCar", based on classic AUTOSAR architecture. Different ADAS features can be tested within this demonstrator by implementing and integrating required software components and modules.

The "TUCminiCar" is developed using an RC racing toy car. This small car uses the standardized AUTOSAR method and helps to test and validate features without the high cost of a full-scale car. With the implementation of an AUTOSAR-based ECU, different sensors from automotive industries, and a custom power supply mechanism, it acts as a real-world vehicle. A remote control can control the car, but to test the ECU-

Tester communication, a CAN communication mechanism is also present within the car ECU. The car uses a Brushed DC motor to control its speed, which is responsible for the smooth movement of the TUCminiCar, allowing for both acceleration and

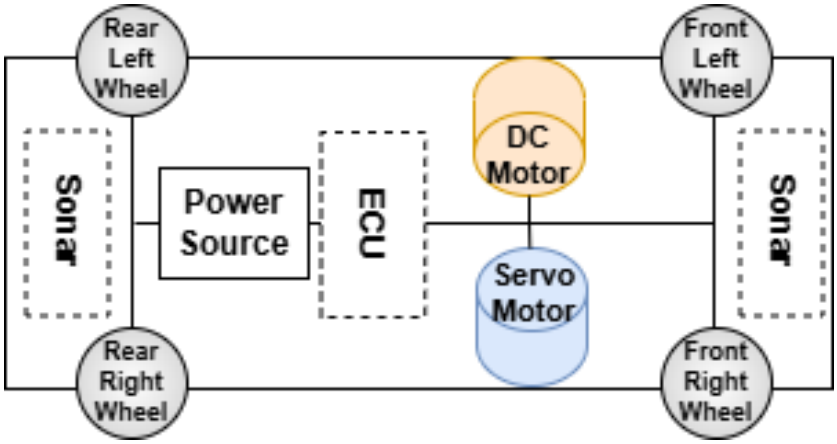

Figure 2.3: TUCminiCar Extract.

deceleration. The ECU can control the motor's output power by using pulse-width modulation (PWM), which adjusts the car's speed. PWM signals facilitate smooth adjustments for different speeds by varying the duration of voltage supplied to the motor. This signal also increases energy efficiency, which is crucial for battery-powered systems since it reduces heat generation. [11] The motor-controlling PWM signal depends on the duty cycle provided by the ECU, which varies from 0 to 200, where 0 to 99 duty cycle acts as reverse speed, 100 is neutral, and 101 to 200 as forward speed control.

Figure 2.4: TUCminiCar DC Motor.

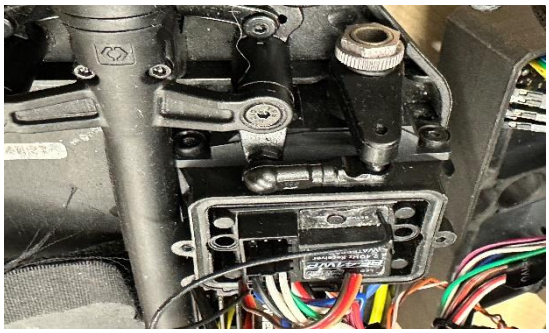

Figure 2.5: TUCminiCar Servo Motor.

For steering control of the car, the demonstrator uses a servo motor to control its steering angle. Like the DC motor, the servo motor is also controlled by PWM sent from the ECU. The servo angle response is read by duty cycle from 0 to 200, where 0 stands for maximum right direction, 100 is neutral, and 200 is for maximum left direction.

The demonstrator has been implemented with an AUTOSAR-supported ECU based on the Infineon TC387 TriCore microcontroller to control the actuators of the car. The TC387 microcontroller is part of Infineon's AURIX family and integrated into the eval

board.[12] Its tri-core architecture and high processing power are necessary to analyze data in real-time for the embedded world. This KIT is designed to support automotive

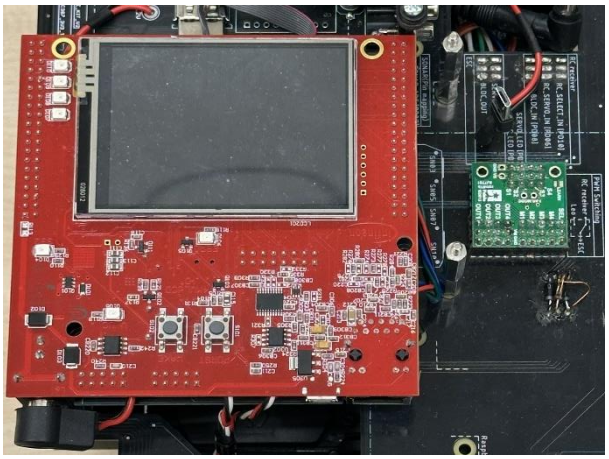

Figure 2.6: KIT_A2G_TC387_3V3_TFT Eval Board on TUCminiCar.

applications requiring high computational power. It provides up to 8 MB of flash memory for different programs and data flash. It has a high-speed CAN transceiver to facilitate a CAN communication interface.[12] There are two 40-pin connectors with the input/output signals available within the board, which can be used to operate different actuators and sensors. Using AUTOSAR classic architecture within this AURIX board, the demonstrator has pre-configured all the basic software components, communication channels, and sensors to perform ADAS activity.

TUCminiCar is also integrated with different automotive sensors to facilitate ADAS functionality. Six sonar (Sound Navigation and Ranging) sensors are implemented within the car to check the obstacle status in the front, back, and sides. The sonar sensor works based on the deflected sound wave on the object staying within its path.

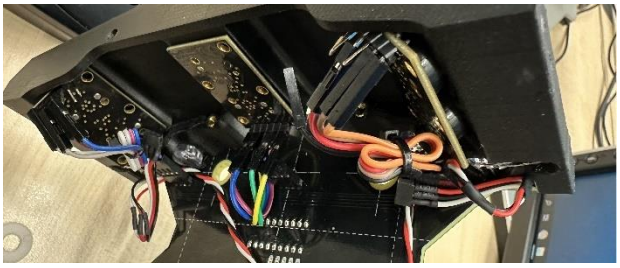

Figure 2.7: TUCminiCar Sonar Sensor.

The sensor has a transceiver that sends sound wave beams continuously; this sound wave travels within a straight line until it gets reflected. After it hits any object, the transceiver will detect the reflected echo of the sound wave again. The object's distance can then be calculated using the time required for the sound wave to travel.[13] The sonar sensor used in the demonstrator has a 2.5-meter detection range coverage. The advantage of using sonar in the demonstrator is that no matter the light condition during the operation, it can detect whether it is bright or

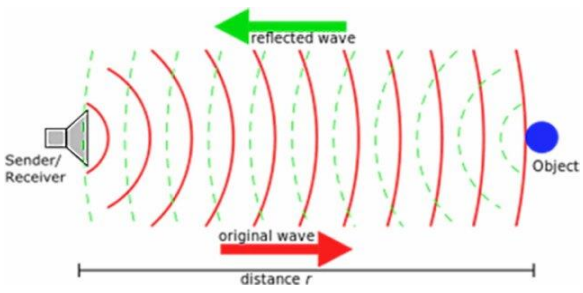

Figure 2.8: Sonar Working Method.[13]

dark. With the straight driving functionality of a car, it is necessary to check the obstacle status while driving to maintain a braking mechanism depending on the road condition.

Apart from the safety-critical components used inside the demonstrator, additional visual and auditory actuators are also used. For the forward drive, two types of light indicators are available: low and high beam. Depending on the requirement, both lights can be activated. There are also indicator lights for reverse driving. It also provides left—and right-side indication lights, which can be used depending on the steering

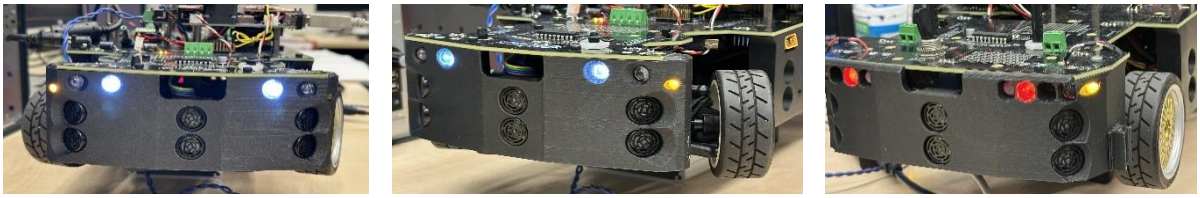

Figure 2.9: TUCminiCar Visual Actuators.

angle configuration. For braking purposes, a pair of red lights are installed on the back of the car. This car also provides a beep-controlling mechanism; this beep control can be activated depending on the situation, like obstacle detection. All these components, already defined within the AUTOSAR architecture's basic software layer, can be controlled from ECU with the developed software component for the atomic straight driving application software component.

2.3 Atomic Straight Driving Core Concept

"Atomic" originates from the word "Atom," defined as the smallest unit of a substance. In computer science, it can be referred to as the smallest section of operation performed without interruptions.[3] For the straight drive concept, atomic is the term where the smallest section can be covered by a vehicle without interruption. The wheel rotation coverage must be defined first to drive the "TUCminiCar" demonstrator in an atomic driving concept. For this purpose, the wheel circumferences must be calculated first. After observation, it was found that each car wheel had a diameter of approx—64mm. According to mathematical formulation to calculate the circumference of a circle, circumference = $pi * d$, Here, $d = 64$ mm, and pi has a constant value of 3.14. From the calculated value, the circumference of the car wheel is 200.96 mm, which can be floored to 200 mm. So, according to the calculation, if the wheel rotates once, either forward or backward, the distance coverage for the car

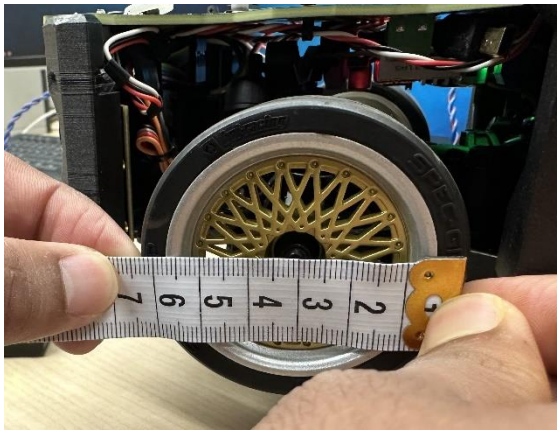

Figure 2.10: TUCminiCar Wheel Diameter.

should be 200 mm. To verify the result, the car was manually rotated once the cycle of wheel rotation was completed, and the result was the same as the calculation.

Based on the wheel circumference calculation and one cycle of wheel rotation, the smallest section length of atomic straight driving is 200 mm of path length.

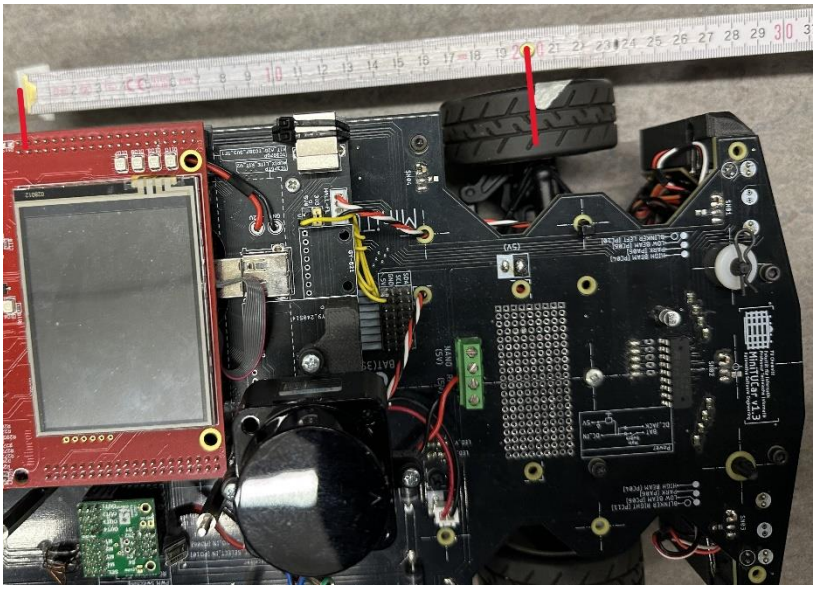

Figure 2.11: TUCminiCar Path Coverage for the One-Wheel Rotation.[15]

Therefore, the initial goal of the driving mechanism is to define an automated braking mechanism needs to be defined so that the demonstrator car can stop automatically after running one of the smallest sections of a straight drive path. This thesis aims to test this shortest path distance by one wheel rotation, including five sections of this small covered section with different speed values. The figure below represents the atomic straight drive concept, where five sections of 200 mm are defined for a straight

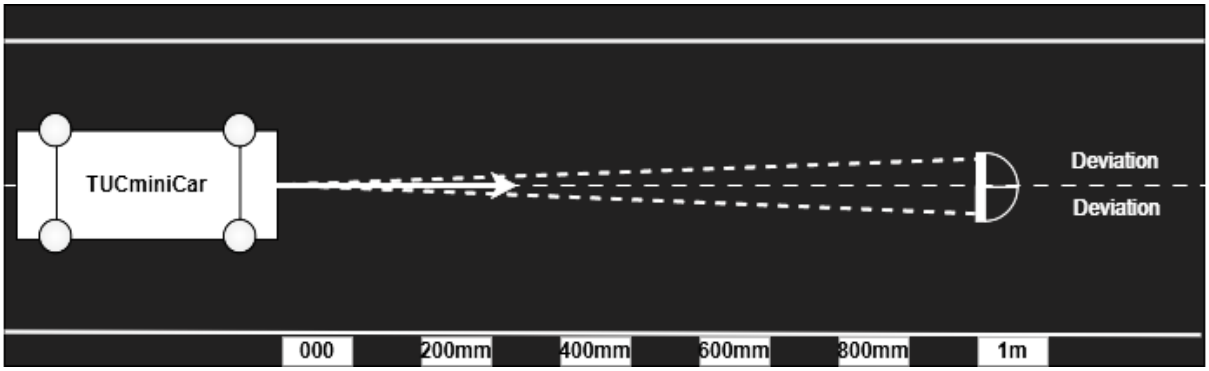

Figure 2.12: Atomic Straight Drive Pattern.

drive path. The total length of the path is one meter. During the straight driving operation, the steering angle should be in the neutral position so that the steering servo can generate a straight path driving angle for the car. The deviation tolerance is kept due to mechanical deficiencies. Deviation tolerance should be maintained ±0.5° to ±3° to acquire greater accuracy.[14] However, the angle deviation depends on the car's

speed and the road's type. In addition to straight driving, another safety critical parameter, which is obstacle detection and taking necessary action after detection, also needs to be tested.

2.4 Development and Testing Environment

Developing and testing software components within the AUTOSAR architecture requires complex devices and software tools. The computer engineering department of TU Chemnitz provides its own AUTOSAR toolchain for standard development procedures in the automotive domain. The figure below illustrates the classic AUTOSAR toolchain from TU Chemnitz. After that, some brief details for each part of the toolchain are described.

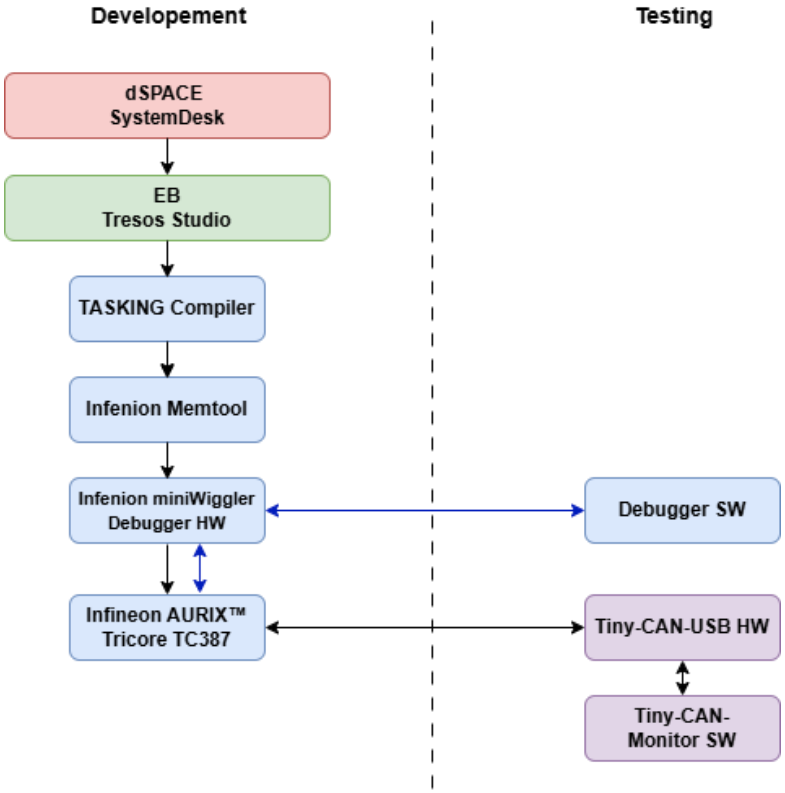

Figure 2.13: AUTOSAR Classic Toolchain.

2.4.1 dSPACE SystemDesk

SystemDesk from dSPACE provides a platform for creating software components (SWC) within the AUTOSAR architecture. These can then be validated and simulated within the platform as well. It is the ideal foundation for checking the software-in-loop (SIL) process. [15] SystemDesk provides the facility to create virtual ECUs and testing environments in virtual simulation (VEOS).[15] It offers several options for working with the development process within AUTOSAR architecture. Since it has built-in AUTOSAR facilities, the development procedure can be started from scratch or

modified by any developed system. It can be used for software component creations, whether the created component is for the basic software (BSW) layer or in the application layer. To work with the existing system file, the system description *.sdp file is required to modify the system. After necessary modification from SystemDesk, the *.arxml file must be generated again to integrate within the system. The Department of Computer Engineering of TU Chemnitz provides facilities for its students to access SystemDesk tools from the Automotive Software Engineering Laboratory to work with the AUTOSAR software components.

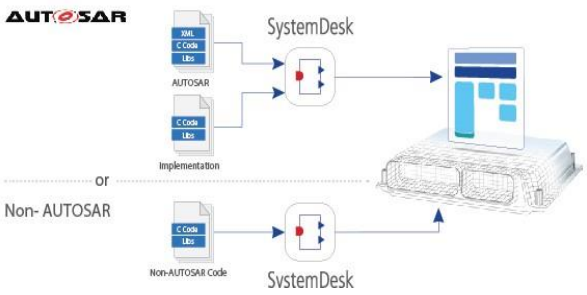

Figure 2.14: dSPACE SystemDesk. [15]

2.4.2 EB Tresos Studio

Elektrobit (EB) tresos studio is the classic AUTOSAR development tooling. EB Tresos studio is fully compatible with the AUTOSAR workflow and can be integrated into existing toolchains.[16] It provides graphical interfaces to configure different AUTOSAR modules for the AUTOSAR stack, whether the BSW or microcontroller abstraction layer (MCAL). It also generates an RTE connection to provide connectivity facilities from the BSW layer to the application layer's software components.

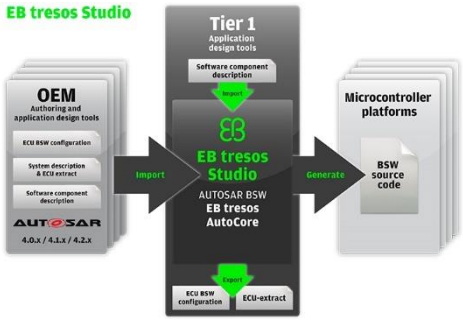

Figure 2.15: EB Tresos Studio. [16]

It offers an extensive predefined library of configuration modules by which developers can configure them in a quick timeline. It provides different module configuration support for the system, from different applications to communication to safety-related modules of AUTOSAR architecture. The EB Tresos studio works in many automotive applications for different manufacturers. It can manage simple systems with a single ECU and more complex systems that utilize multiple ECUs for various ADAS functionalities and autonomous driving. The tool offers configurations for automotive diagnostics, security, and communication protocols, making it suitable for safety-critical automotive applications. The (.arxml) file generated from SystemDesk needs to be imported into Tresos Studio, and then the project must be generated after verification to compile it. Automotive Software Engineering students of TU Chemnitz also have access to this software tool in the lab.

2.4.3 Infineon Compiler and MemTool

The “TASKING” compiler from Infineon is mostly used by developers worldwide in the automotive domain. High-performance tools are required for safety-critical embedded systems, especially in applications that require strict industry standards and reliability. TASKING compilers support various microcontroller architectures, including the Infineon AURIX™ TriCore microcontroller TC387 used in the ADAS demonstrator TUCminiCar.[17] TASKING is known for its excellent code optimization skills, allowing it to create effective code that uses less memory and power, and these are essential requirements in embedded systems. It offers runtime libraries and safety-critical compilers to meet industry safety standards for automotive functional safety (ISO 26262).[18] TASKING compilers support AUTOSAR for automotive applications, providing developers with the necessary tools to develop, configure, and test applications in frameworks that comply with AUTOSAR. The program generated by EB Tresos Studio needs to be compiled to generate a *.hex file.

Figure 2.16: TASKING Compiler.[18]

Like the TASKING compiler, Infineon MemTool is application software for flashing data developed by Infineon Technologies. This tool enables users to program, erase, and manage memory in Infineon microcontrollers. The MemTool offers a graphical user interface to erase memory, flash the generated *.hex file from the compiler, and verify it with this tool.

2.4.4 Debugger Hardware and Software

The Infineon DAP miniWiggler is a debugging interface for Infineon microcontrollers. It is used for flashing software, reading memory, and accessing memory locations. It converts from the PC/USB to an Infineon Microcontroller device's debug interface (10-pin DAP).[19] This hardware is required to connect and flash the *.hex file from the mem tool software to the ECU. It is like providing a bridge between the microcontroller and a tester (computer), supporting communication protocols like DAP (Device Access Port), which are essential for debugging and memory access. For the

Figure 2.17: Infineon DAP miniWiggler V3.

software side, the AURIX™ command-line debugger tool can be used as a debugger software. This is an open-source software that operates to access read memory location data.

2.4.5 Tiny-CAN Hardware and Software

Tiny CAN is a USB-to-CAN interface device that connects a tester (PC) to a CAN (Controller Area Network) bus system. It allows developers to test different data by communicating with the ECU from their computer. It is very useful for establishing reliable tester-ECU communication in the automotive domain. It provides a real-time data transfer facility with a transfer rate of up to 1 MBit/s, and the data log can be saved for later analysis. Tiny-CAN complies with the ISO11898-2 standard, which is a standardized CAN technology. [20]

Figure 2.18: Tiny-CAN II-XL Interface

The Tiny CAN Monitor is a graphical application software tool with Tiny CAN hardware interfaces. It enables the monitoring, analysis, and debugging of CAN (Controller Area Network) messages. The graphical interface supports different operating systems. This

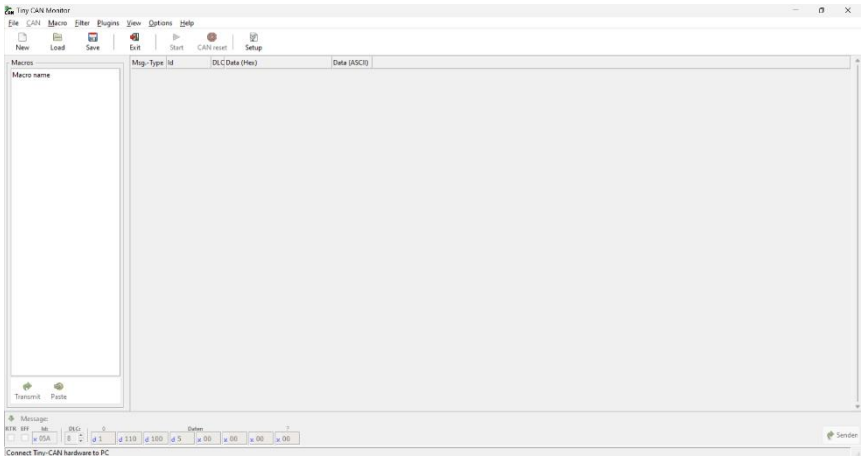

Figure 2.19: Tiny-CAN Monitor

allows the data filter to be defined as having specific CAN ID data. By defining the data length code (DLC), it is also possible to send data byte by byte. Different data parameters, like speed, distance, steering angle, etc., can be sent to the TUCminicar from the tester to test the straight drive functionality.

3 State of the Art

The complexity and additionality of vehicle features are growing fast, leading to a need for scalable, reusable software that works across different hardware. This has increased the development of automotive software components within a defined standard. AUTOSAR, first introduced in 2003, stands at the core of this evolution as an initiative toward standardization in software architecture for the automotive domain. With the advancement of driverless vehicle technology, it becomes more necessary to implement different software components that will assist the vehicles in fulfilling ADAS functionality within the AUTOSAR framework. The thesis will try to find answers to the following work scopes:

- Analyze AUTOSAR within an ADAS Demonstrator
- Develop SWC to Maintain Atomic Straight Drive within AUTOSAR
- Test ADAS functionality

So, reviewing and gathering scientific knowledge from the current development processes within these work scopes is necessary. The following review will assess the “State of the Art” in developing an AUTOSAR SWC for ADAS functionality. It will provide an overview of AUTOSAR's role within automotive systems, including the tools and methodologies for development and simulation and how seamless integration among classic AUTOSAR software components is achieved. The state-of-the-art chapter is divided into five subsections: “Current Trends and Approaches (3.1)”, “Comparative Analysis (3.2)”, and “Relevance to the Thesis and Gap Analysis (3.3)”; these three sub-chapters will have some valuable insight into ongoing AUTOSAR related works, the next one “Adaptive User Interface for Automotive Demonstrator (3.4)” will have some discussion over non AUTOSAR development process and finally based on all gathered knowledge the last part “Proposed Work (3.5)” to have a clear observation step by step.

3.1 Current Trends and Approaches

This section emphasizes the following three significant related papers, dividing them into three sub-sections to provide information on recent advances in AUTOSAR-based systems and their applications.

First, "Dynamic Architectural Simulation Model of YellowCar in MATLAB/Simulink Using AUTOSAR System"[21] provides an overview of a MATLAB/Simulink-based simulation framework for integrating AUTOSAR and non-AUTOSAR ECUs. This

demonstrates the flexibility of modular simulation tools in iterative automotive software development.

The second one, "Modeling and Development of AUTOSAR Software Components," [2] presents the development of reusable and resource-efficient software components, considering the layered AUTOSAR architecture, VFB, and RTE for real-time communication.

Finally, "Design and Implementation Procedure for an Advanced Driver Assistance System Based on an Open Source AUTOSAR" [22] describes the use of open-source tools in implementing ADAS. It outlines how to realize AUTOSAR-compliant communication and sensor integration.

In addition, another paper, "Adaptive User Interface for Automotive Demonstrator, " from a non-AUTOSAR implementation, was also reviewed. [23] Since it was based on the TUC demonstrator's simulation, which uses CAN-Bus, it can provide some reliable data regarding this. This paper will be analyzed separately in the (3.4) chapter since it is not fully AUTOSAR-based rather than using non-AUTOSAR.

3.1.1 Dynamic Architectural Simulation Model of YellowCar in MATLAB/ Simulink Using AUTOSAR System

The YellowCar demonstrator at the "Chemnitz University of Technology" is an innovative research and educational tool from the computer engineering professorship for advanced automotive technologies. It is the first demonstrator built for the Automotive Software Engineering lab based on AUTOSAR 2.1 architecture.[24] The YellowCar has been designed and integrated with different features, including 3 AUTOSAR-based ECUs, the CAN bus for communication purposes between ECU-

Figure 3.1: YellowCar Picture.[21]

tester, and ultrasonic sensors for obstacle detection. These features enabled the car to create a simulated scenario similar to real-world applications like parking assistance and autonomous driving. Based on this setup, the paper "Dynamic Architectural Simulation Model of YellowCar in MATLAB/Simulink Using AUTOSAR System"[21] elaborates on how principles of AUTOSAR integrate with MATLAB/ Simulink for the simulation and testing of YellowCar. The key findings which have been illustrated in the thesis are as follows:

- **The Purpose of Simulation**

There have been concerns over the requirement for simulation in the automotive research area to check compatibility for different SWCs within real-world scenarios. For real-world implementation, it is necessary to validate systems and functionalities testing without having a real-world hardware setup. YellowCar is an adaptable, scalable platform for modeling autonomous and semi-autonomous systems.

- **Integration within AUTOSAR**

The AUTOSAR framework defines a modular, scalable, and hardware-independent software architecture. This standardization allows developers to implement Software Components (SWCs) and simulate their system behavior dynamically.

- **MATLAB/Simulink Modeling**

In this paper, MATLAB/Simulink is used as the main simulation environment, where the architectural models of the YellowCar are developed. The environment supports Model-in-the-Loop (MIL) Simulation of AUTOSAR and non-AUTOSAR components. It demonstrates the simulation of components such as ECUs, sensors, actuators, CAN, RTE, and Virtual Functional Bus (VFB).

- **Advantages of Simulation**

Accuracy: The simulations replicate real-world behavior of simple and complex driving patterns, such as straight driving or avoiding obstacles.

Cost-Effectiveness: Saves the need for frequent hardware upgrades since a simulation-based validation can be done.

Scalability: Supports integration of additional ECUs, sensors, or software modules without re-designing the system.

Standardization: Adapted to the AUTOSAR framework ensures portability and interoperability across different platforms and vehicles.

Flexibility: Allows hybrid testing of AUTOSAR and non-AUTOSAR components.

- **Disadvantages of Simulation**

Simulation Limitations: Some real-world components may not fit well since it is a simulation environment.

Cost of Tools: While cost-effective in the long run, tools like MATLAB/Simulink with AUTOSAR support can have high licensing fees.

3.1.2 Modeling and Development of AUTOSAR Software Components

The paper "Modeling and Development of AUTOSAR Software Components"[2] systematically discusses the systematic methodology for designing, configuring, and implementing AUTOSAR software components. The implementation starts with defining system inputs, known as the System Configuration Input, which includes selecting software components and hardware and establishing architectural definitions. AUTOSAR facilitates this with architectural templates for validating initial designs.

Next, the Configuration Description maps software components to Electronic Control Units (ECUs) while arranging network topologies and bus mappings. The following

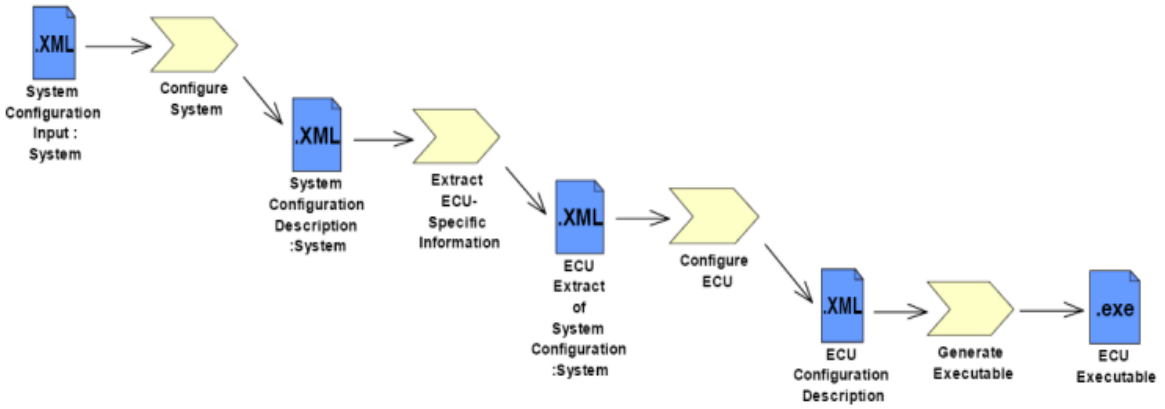

Figure 3.2: Overview AUTOSAR Methodology. [2]

steps extract and refine specific ECU information, generating an ECU Configuration Description. This description includes configurations for Basic Software (BSW), task scheduling, and the assignment of runnable entities. Finally, this will result in the compilation and linking of code into an executable. After having a thorough analysis of the paper, the following key points have been identified:

- **Purpose**

This paper tries to provide a structured way of developing AUTOSAR software components, which considers the overall development process, from modeling to source code generation. This shall ease and standardize the development of automotive application software in a context where the increased complexity of the software and hardware dependencies causes challenges.

- **Integration of AUTOSAR Software Components**

AUTOSAR has systematic component integration, which is made by defining system configurations, defining interface types, mapping software components to ECUs, and configuring BSW. The use of standardized templates and tools facilitates smooth integration, ensuring modular and reusable components.

- **Software Component Template**

Software Component Template describes the main aspects of AUTOSAR component development:

Internal Behavior: Describes the internal behavior of a Software Component that is composed of Runnable Entities and their responses to RTE events.
Runnable Entities are the smallest, schedulable units of code written in a programming language, such as C, and triggered by RTE events or BSW schedulers.

Runnable Entities: Code units that can be scheduled, developed in C, and triggered by Run Time Environment (RTE) or Basic Software (BSW) schedulers.

RTE Events: Events that perform runnable, including timing, mode switches, etc.

Contract Phase: RTE-specific APIs are generated from component descriptions to ensure the software's independence from the communication implementations.

Deliverable: Final products include SWC-type descriptions, internal behavior details, and implementations for system generation.

3.1.3 Design and Implementation Procedure for an Advanced Driver Assistance System Based on an Open Source AUTOSAR

The paper "Design and Implementation Procedure for Advanced Driver Assistance System Based on an Open Source AUTOSAR" [22] describes the integration and development procedure for advanced driver-assistance systems based on an open-source AUTOSAR platform. Open-source tools can be used to design a low-cost, flexible platform for ADAS implementation, bringing down the development cost while keeping up with the AUTOSAR standard. This paper uses the real-time obstacle avoidance experiment to prove this method's capability, mainly in vehicle safety and

reliability improvement. Based on the paper review, the following key points have been discovered:

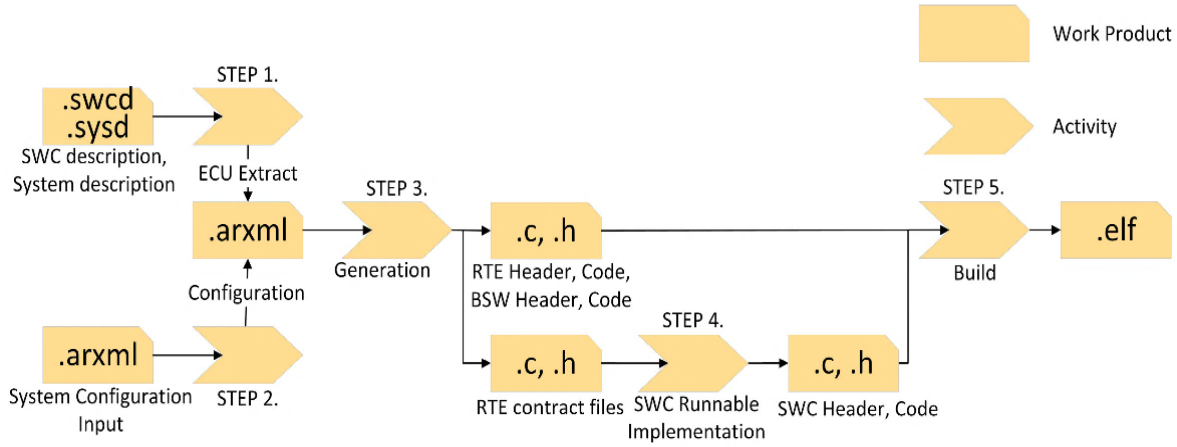

Figure 3.3: Open source AUTOSAR Procedure. [22]

- **Purpose**

The paper demonstrates how open-source AUTOSAR frameworks might be used to design and implement ADAS. Hence, it provides a systematic way of developing safety-critical automotive features like collision warning systems. This study uses open AUTOSAR, which would reduce dependence on proprietary solutions and make it much more cost-efficient. It also shows the possibility of integrating AUTOSAR-compliant, modular, and scalable software components that ensure interoperability and portability to different hardware and software configurations.

- **Open Source AUTOSAR**

It focuses on Open AUTOSAR, an alternative open-source to expensive commercial software systems. The research highlights that Open AUTOSAR can standardize software development while still retaining the possibility of developers creating reusable and modular software components. Open-source AUTOSAR also provides interoperability features necessary for manufacturers and developers working with different vehicle platforms and ECUs. The paper tries to demonstrate how the availability of open-source frameworks empowers small developers to adopt industry-standard technologies, improving innovation within automotive software development.

- **ADAS Collision Warning**

This paper presents a collision warning system as an example of how Advanced Driver Assistance Systems (ADAS) work within the AUTOSAR framework. This warning system processes data from different sensors and predicts possible collisions. It relies on Runnable Entities, the smallest schedulable code units in AUTOSAR. These entities

are activated by Runtime Environment events initiated by sensor inputs. The paper points out that the system is designed for low latency and high reliability at the center of real-time decision-making within collision warning systems. This implementation also demonstrates the scalability of open-source AUTOSAR in integrating more ADAS functionalities in complex situations.

- **CAN Communication**

This paper also provides information about integrating the CAN (Controller Area Network) communication protocols in the ADAS framework. CAN communication is very useful and widely used for real-time data exchange between different ECUs in the system. AUTOSAR simplifies this by providing predefined standardized templates for CAN message formats, allowing the ADAS system to respond promptly to various driving conditions. Reliability makes CAN communication the number one choice for safety-critical applications such as collision avoidance, as it allows the system to operate under various conditions without failure.

3.2 Comparative Analysis of Current Trends and Approaches

The three papers that have been studied contribute to AUTOSAR-based system development from clearly different yet interrelated perspectives. Each highlights modularity, standardization, and flexibility within automotive software. The first addresses dynamic architectural modeling with MATLAB/Simulink; the second outlines concepts and methodologies for AUTOSAR; the other one targets advanced driver-assistance systems (ADAS), especially collision warning, through open-source frameworks. Here's a comparative table based on the three papers in terms of working with the thesis topic "AUTOSAR Software Component for Atomic Straight Driving Pattern":

Table 3.1: Comparison of Current Trends and Approaches in AUTOSAR.

Research / Aspect	Dynamic Architectural Simulation Model of YellowCar in MATLAB/Simulink Using AUTOSAR System	Modeling and Development of AUTOSAR Software Components	Design and Implementation Procedure for an Advanced Driver Assistance System Based on an Open Source AUTOSAR
Purpose	Simulate automotive components in MATLAB/Simulink to validate AUTOSAR integration.	Standardize the development of AUTOSAR software components, focusing on modularity and reusability.	Develop ADAS functionalities using an open-source AUTOSAR framework, addressing the collision warning method.
Common Features	Uses AUTOSAR for modularity and standardized development.	Adheres to AUTOSAR principles for layered architecture and reusability.	Employs AUTOSAR-based methods for real-time communication and system efficiency.
Knowledge Acquired	Simulation of AUTOSAR components	AUTOSAR Software Components (SWC) creation process	Implementation of ADAS feature in ECU
Limitations	Limited to simulation; requires further work for real-world deployment.	High complexity and resource-intensive due to detailed configurations and template management.	Lacks the robustness and support of proprietary AUTOSAR tools for commercial scalability.
Main Undisclosed Area	Worked with simulation only, no concrete idea to deploy in real-world systems	No practical applications have been demonstrated.	Due to the open-source AUTOSAR platform used, no proper information to migrate to the classic AUTOSAR

3.3 Relevancy to the Thesis Topic and Gap Analysis

The presented research papers provide all the necessary knowledge for a foundational and practical understanding of developing an AUTOSAR software component that realizes the atomic straight driving pattern. The following section shows the relevance and gap in the deployment of the software component for atomic straight driving.

3.3.1 Relevancy

The findings from the papers put together projects on modularity, simulation, and resource-efficient design in the development of AUTOSAR software components. Structured development and reusability of software components ensure the efficient handling of straight-driving tasks with consistency in communication and interaction

across system components. Relevancy information for each of the paper analyses is provided below.

Dynamic Architectural Simulation Model of YellowCar in MATLAB/Simulink Using AUTOSAR System[21]**:** This paper is directly relevant since it presents how AUTOSAR can integrate with simulation tools, like MATLAB/Simulink, to provide insights into architectural modeling and validation. The methods used for the simulation of vehicle dynamics can be instructive for modeling and testing atomic straight driving patterns in a controlled environment.

Modeling and Development of AUTOSAR Software Components[2]**:**
This paper provides a detailed explanation of the AUTOSAR methodology, which emphasizes modular development of software components. The software logic necessary for handling straight-driving tasks efficiently with reduced resource allocation is developed by focusing on strong reusability and well-structured communication via the Runtime Environment (RTE).

Design and Implementation Procedure for an Advanced Driver Assistance System Based on Open Source AUTOSAR[22]**:** Although this paper is not directly relevant due to the use of open-source AUTOSAR platforms, ideas are still gathered for reducing development costs and experimenting with different driving patterns. The methodology of integrating sensors and visual actuators into AUTOSAR-based systems is also helpful in detecting and maintaining ADAS functionality.

3.3.2 Gap Analysis

While these papers offer a strong basis for understanding AUTOSAR software components and their applications, specific gaps related to the thesis topic can be identified:

Dynamic Architectural Simulation Model of YellowCar in MATLAB/Simulink Using AUTOSAR System[21]**:**
Influence: Research highlights the dynamic simulation of AUTOSAR systems, which is relevant for testing software components (SWC) of atomic driving patterns in simulation.

Gap: Emphasizes architectural modeling but does not apply it to specific driving scenarios such as atomic straight driving.

Modeling and Development of AUTOSAR Software Components[2]**:**
Influence: Research focuses on comprehensive coverage of AUTOSAR methodologies, software templates, and RTE communication, which are essential to designing atomic driving patterns within AUTOSAR architecture.

Gap: It does not address specific driving tasks or patterns and only provides generic guidelines for software component development.

Design and Implementation Procedure for an Advanced Driver Assistance System Based on Open Source AUTOSAR[22]**:**
Influence: Research demonstrates practical applications of AUTOSAR in ADAS, real-world integration of sensors, and control mechanisms.

Gap: This research is mainly focused on the ADAS and collision avoidance functionality with the open-source AUTOSAR platform and does not provide any idea regarding the deployment strategies of an atomic driving pattern.

To summarize, the above analysis of research papers proves their relevance in understanding the AUTOSAR software components in automotive systems. However, they missed the atomic driving pattern focus using AUTOSAR, which creates an opportunity to bridge the gap in this thesis. In the next section, another paper's review is conducted, which is non-AUTOSAR-based and thus analyzed separately in a separate sub-chapter (3.4).

3.4 Adaptive User Interface for Automotive Demonstrator

The main goal of the research work "Adaptive User Interface for Automotive Demonstrator"[23] was to make an adaptive system where users can interact with the Application Programming Interface (API) instead of going through source code. The Adaptive User Interface (AUI) enhances the functionality of platforms like the BlackPearl demonstrator at TU Chemnitz by enabling dynamic interactions, such as touch, voice commands, and remote access. This paper provides valuable ideas on ongoing research works with the ADAS demonstrator in TU Chemnitz. In this research work, CAN-Bus is used as a crucial part of the communication network within the demonstrator BlackPearl. This is a system to provide valid data interchange in real-time between the interfacing units. It provides a way for the CE-Box and the different Raspberry Pi boards to exchange important information concerning sensor and actuator data. CAN-Bus communication supports scalability, which makes it easy for different hardware servers to join the network easily. It ensures minimal latency and

high performance, even with various input components like cameras, streaming dashboards, and remote commands.

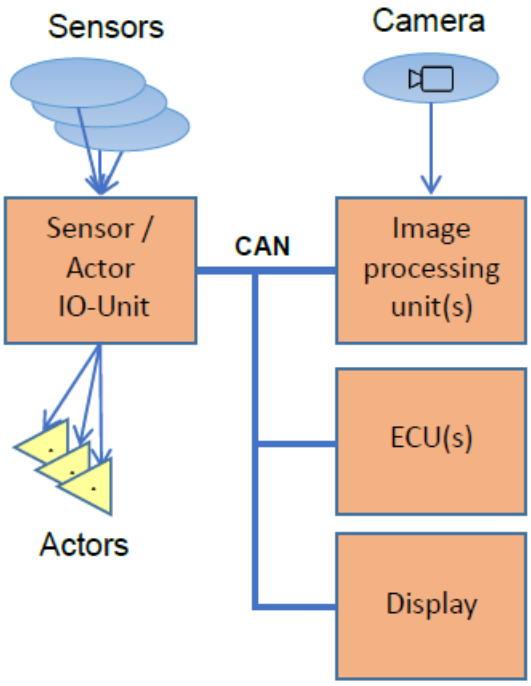

Figure 3.4: Modules of the BlackPearl Demonstrator. [23]

Specific rules for reading or writing CAN-Bus messages ensure their reliability. This research explains how the CAN bus is integrated among different sensors within a demonstrator. Since CAN communication will be used in the thesis topic, “AUTOSAR Software Component for Atomic Straight Driving Patterns,” the idea of the CAN message format within the existing research will provide a better understanding of this.

3.5 Proposed Work: “AUTOSAR Software Component for Atomic Straight Driving Pattern

This thesis work aims to design an AUTOSAR-compliant software component that realizes the atomic straight driving pattern—a core fundamental building block used in more complex navigation tasks. The dSPACE simulation tool will model the atomic straight driving pattern through incremental movements and interaction simulations. It will then be integrated into the vehicle's ECU to comply with hardware competency validation. It has an obstacle-detection feature for testing ADAS functionalities, ensuring a vehicle can operate safely and efficiently. All this ensures compatibility on AUTOSAR platforms and the ability to perform in real-world scenarios.

4 Methodology

This chapter provides ideas about developing an AUTOSAR software component for the atomic straight driving pattern. It starts with a detailed analysis of the software's requirements to understand the functionality. Then, it describes the implementation of the AUTOSAR Classic to the ADAS demonstrator, showing its importance to the overall system architecture.

This chapter also covers the CAN protocol, developed to communicate from tester to ECU. It also looks at the configuration processes of key development tools such as dSPACE SystemDesk and EB Tresos to understand how to deploy SWC for atomic straight driving. Finally, it provides the development model information for developing software components within AUTOSAR Classic. These are some of the basic building blocks of the software component, which must be accomplished with complete functionality within the system for its integration to be accomplished. This chapter is divided into five basic sub-sections: Development Model: The V-Model (4.1), Requirement Analysis (4.2), Understanding AUTOSAR Classic and Modules (4.3), Understanding Controller Area Network (CAN) (4.4), and Analyze “TUCminiCar” System Configuration (4.5) which will be reviewed in details in the following sections.

4.1 Development Model: The V-Model

The V-Model development process is the most widely used embedded software development, especially when the AUTOSAR framework is involved. It can be described as a software development methodology highlighting sequential, structured approaches to software development. It extends the waterfall model by linking each phase in the development from requirements through design to implementation, correspondingly with unit testing, integration testing, and system testing.[25] Each phase of this model represents ongoing validation and verification to ensure quality and traceability. In other words, it is especially suitable for safety-critical systems.

4.1.1 Overview of the V-Model

The V-Model is known as the Verification and Validation Model.[26] This software development framework focuses on a structured and traceable approach toward designing, implementing, and testing systems. Its defining characteristic is its "V" shape, where the left-hand side of the "V" represents requirement analysis and designing of high and low-level systems while the right-hand side is used for different integration and testing purposes. With every development phase, there is also a testing phase to cross-check. The steps of a model can be planned according to software

development requirements. This V-Model is very much applicable to the thesis "AUTOSAR Software Component for Atomic Straight Driving Patterns," as this model is used in bringing the entire process of developing atomic straight driving pattern functionality properly and giving it extensive testing in combination with other system components like CAN communication and ADAS features.

4.1.2 Mapping V-Model to AUTOSAR Development

In AUTOSAR development, the V-model supports traceability, modular design, and validation throughout the entire software life cycle. Mapping the V-model phases into AUTOSAR development improves the quality and productivity of automotive software. Since the thesis is based on implementing SWC within the AUTOSAR framework, a V-model is necessary to maintain the work process in a standard manner. Below is the designed V-model for this research project's development life cycle.

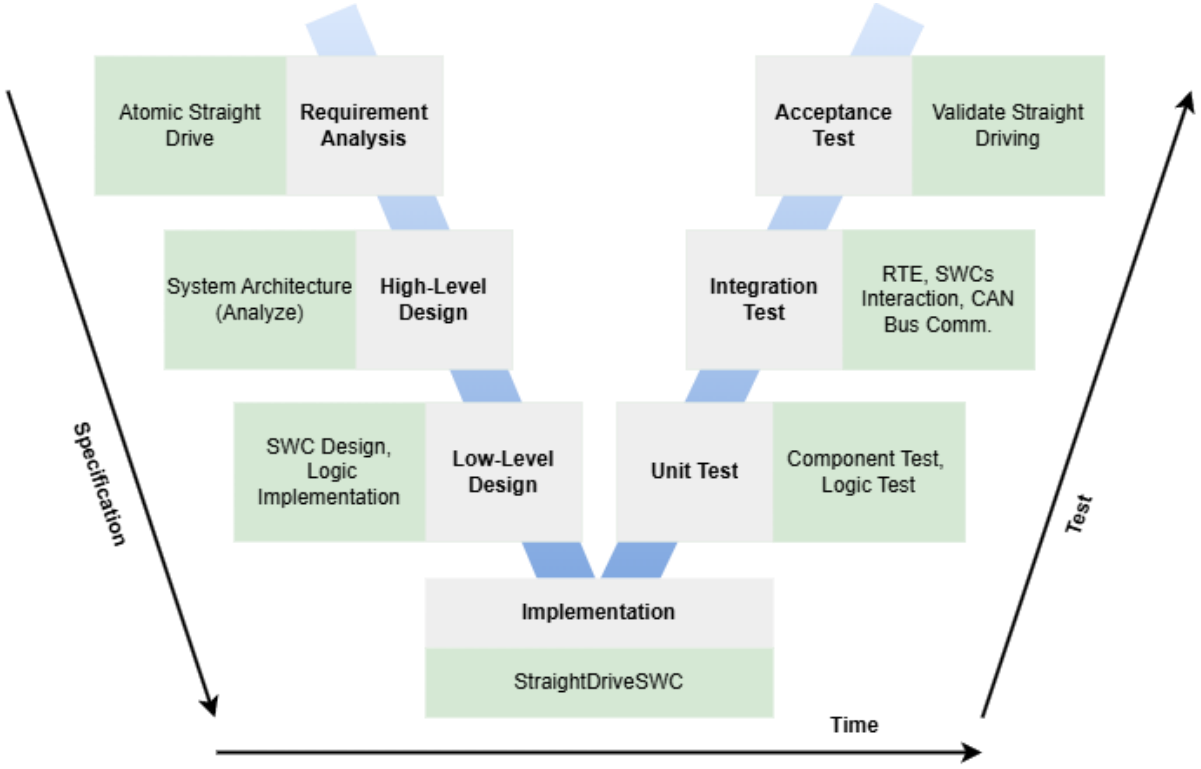

Figure 4.1: V-Model Development for Atomic Straight Drive SWC.

The left side of the V will unveil all the requirements and design procedures. The right side represents all testing and acceptance, with implementation placed in between. The below steps have been illustrated in the developed model:

Requirement Analysis: The V-Model begins with the Requirement Analysis step in AUTOSAR development. This involves defining the functional and non-functional

requirements of the atomic straight driving pattern. The methodology chapter will discuss this level of explanation in detail.

High-Level Design: This level defines the overall system, including architecture and communication methodology. It requires a System Architecture Overview since the TUCminiCar is already within classic AUTOSAR architecture, which needs to be reviewed for further development within this demonstrator. Understanding AUTOSAR classic and CAN will also be discussed in detail within this methodology chapter.

Low-Level Design: The leftmost lower step in the V-Model is the Low-level design. This step focuses on SWC Design and Obstacle Detection logic design. Since SWC will be developed during the implementation phase, the existing design in dSPACE SystemDesk for the TUCminiCar and module configuration in EB Tresos studio need to be analyzed first in this step. This analysis will be done within the methodology chapter.

Implementation: This phase works as a bridge for the left side of the model with the right side where testing is placed. In terms of the thesis project, this is the coding of the StraightDriveSWC that will meet the functional and non-functional requirements defined before. Development of the StraightDriveSWC logic in AUTOSAR using dSPACE SystemDesk, importing configuration from dSPACE to EB Tresos Studio, obstacle detection logic, and handling of CAN messages for the system should be defined. A separate chapter is placed after the methodology for the Implementation process.

Unit Test: The rightmost side of the model starts with the Unit test. In this phase, the developed SWC will be tested within the simulation environment (SIL) by creating a virtual ECU in dSPACE and also in the hardware. Different logic developed within the SWC will be tested at this phase.

Integration Test: In this phase, the developed SWC and codes will be integrated into EB Tresos Studio. RTE generation will be verified so that different SWCs' interactions can be tested. After that, hardware setup and CAN communication will be ensured.

Acceptance Test: This final step validates the Atomic Straight Driving. Acceptance tests will be performed by operating a complete system test in the TUCminiCar, where StraightDriveSWC runs to perform various ECU functionalities. Scenarios include straight driving under different conditions, facing obstacles during driving, and sending/responding to CAN messages.

4.1.3 Benefits of Using V-Model:

The V-Model has many benefits for the development life cycles, which make it a suitable model for developing and validating the AUTOSAR Software Component for Atomic Straight Driving Patterns. The main advantages are listed below:

Traceable and structured: The V-Model is a structured development process in which all phases are separated. Every design-related task on the model's left side has a corresponding testing phase on the right side, ensuring traceability of the requirements from start to finish.

Early Detection: Aligning the development stages with specific test activities helps to detect errors early during requirements verification and design validation, which reduces the cost and complexity.

Modularity and Reusability: This model supports modular design and complies with AUTOSAR's principles of reusability software components. This approach makes development easier and supports scalability for future projects for the SWC development.

Safety and Reliability: The structured testing phases of the V-Model are very important for safety-critical systems such as automotive software development. For this project, the integration and acceptance test conducted in the model for AUTOSAR SWC for atomic straight drive ensures that the required driving functionality works correctly within the system.

Adaptability for Embedded Systems: V-Model, focusing on verification and validation, is especially helpful for embedded systems where software and hardware integration are crucial. This project makes the StraightDriveSWC easily interact with other components, like CAN communication or ADAS functions. Following the V-Model allows a systematic and reliable development process, which ensures operational efficiency, safety, and compliance with automotive standards.

4.2 Requirement Analysis

Requirement analysis is the primary task for any software development process. In this project, it this analysis focuses on understanding and documenting the functional and non-functional requirements of the AUTOSAR software component for atomic straight driving. Atomic straight driving is a small but fundamental driving operation that forms a basis for broader vehicle control, such as the consistent straight driving capability, representing one of the ADAS functionalities. Requirement analysis demands two

major analyses: Functional Analysis and Non-Functional Analysis,[27] described in the subsections below.

4.2.1 Functional Requirements

Functional requirements can be defined as essential requirements of developed systems.[27] A project implementation cannot succeed without fulfilling these requirements, so these are the minimum requirements of a system that must be accomplished. For the thesis topic, atomic straight drive patterns, functional requirements can be defined as below:

Atomic Straight Driving: This requirement is the core of the thesis project. The developed software component (SWC) should control the car's actuators to maintain the atomic straight driving at different speeds and distances.

Obstacle Detection: The SWC should implement obstacle detection logic so that it can read data from detection sensors and process it to control the car after detecting an obstacle.

CAN Communication: The SWC should have a mechanism to ensure CAN communication. It should ensure the tester-ECU communication, taking input from the CAN message and sending it to the corresponding components to operate accordingly.

Runtime Environment (RTE) Integration: The developed SWC must be integrated with the AUTOSAR runtime environment (RTE) to handle the triggered events within AUTOSAR and provide output to corresponding components for further processing.

Testability and Modularity: The system should support simulation-based testing (SIL) and modular updates to advance future extensions of ADAS features.

4.2.2 Non-Functional Requirements:

Non-functional requirement analysis is the minimum requirements that the project needs to achieve.[27] It can be described as a developed project's performance and quality standards. For the thesis topic, non-functional requirements can be defined as below:

Performance: Since it is a safety-critical system, it must show low latency. The response should be real-time for parameters like acceleration, braking, obstacle detection, and all other input-output responses handled by this SWC.

Scalability: The developed SWC must support future extensions for applying more complex ADAS features. This scalable system should be aligned with the AUTOSAR framework.

Reliability: Since the advanced driver assistance system deals with safety-critical operations, the developed system must be reliable to its users. The system should have a backup mechanism that can be activated if the primary method fails.

Resource Optimization: Embedded systems always require optimized resources. This will avoid overloading the ECU based on priority uses. Resource optimization needs to be ensured for the thesis project.

Compliance: Although the project is based on a miniature real-world application, it should still comply with AUTOSAR safety standards, like ISO 26262 functional safety. Which will provide the benefit of adhering to real-world industry requirements.

This analysis ensures that the SWC for the atomic straight driving pattern will be robust and reliable by implementing and complying with functional and non-functional requirements. This will not only meet the project's needs but also those of the industry guidelines. These requirements will also guide the subsequent design, implementation, and testing phases.

4.3 Understanding AUTOSAR Classic

The Technical Background chapter already discusses an overview of the AUTOSAR standard. Since the application software component (SWC) needs to be maintained in the AUTOSAR classic framework, it is necessary to analyze the AUTOSAR classic layer by layer to understand its functionality. The AUTOSAR architecture is mainly built on three software layers: Application, Runtime Environment (RTE), and Basic Software (BSW). The basic software layer can be divided into four layers: Services Layer, ECU Abstraction, Microcontroller Abstraction Layer (MCAL), and Complex Drivers. [10] Different layers serve different purposes. Implementing application software components within an AUTOSAR architecture, built in a demonstrator, must be analyzed first. This will provide a clear idea of how to incorporate it according to requirements. Since the TUCminiCar already provides AUTOSAR Basic Software (BSW) modules integrated within ECU, they must be analyzed for application software component deployment. In the following sub-section, an overview of different layers will be discussed in terms of project requirements.

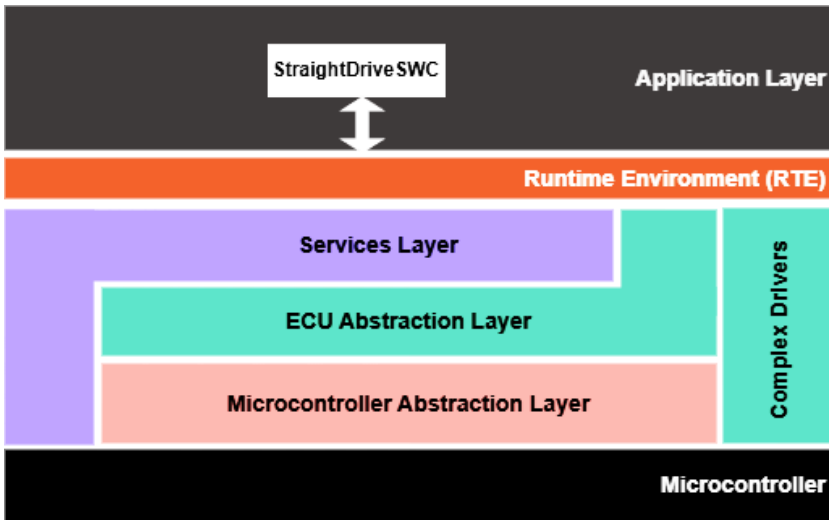

Figure 4.2: AUTOSAR Layered Architecture.

4.3.1 Overview of AUTOSAR Layers

This subchapter analyzes some details about the AUTOSAR layers. It starts with the topmost layer of the architecture, the application, then the runtime environment (RTE), and finally, the basic software layer (BSW).

4.3.1.1 Application layer

The application layer is the topmost layer in the AUTOSAR architecture.[10] It is where the application software component “StraightDriveSWC” for an atomic straight drive for the thesis project should be implemented. Since it will have the SWC, which will implement different control logic, it is the most critical part of the project. Some key features and components belong to the application layer discussed below:

Application Software Components: These are the components that are the building blocks of the application layer. There will be ports for communicating to or from these software components. These ports can use either sender-receiver or client-server interfaces. These ports will control different actuators, such as sensors and motors. These software components will implement the control logic to control those actuators. There will be a runnable within this software component, which will define the component's behavior. It can be triggered by different events, like time triggering, one of which will be used to trigger the “StraightDriveSWC.”

Platform Independence: SWCs in the application layer of the AUTOSAR architecture are platform-independent. That means the application logic implemented within these SWCs is hardware-independent and should work in another hardware platform.

Integration with RTE: The runtime environment (RTE) provides communication services for the SWCs implemented in the application layer. Successful integration will facilitate the application layer's communication with lower-layer modules, like in the basic software (BSW) layer.

Functional Logic: The application layer deals with functional logic deployment and maintenance. For the thesis topic, functional logic must be implemented to control actuators to maintain atomic straight drive and sensor data control to comply with ADAS functionality.

Extensibility for Future Features: The SWCs in the application layer are designed the way these should be applicable to be extensible. If any safety feature is added to the application layer, the currently developed SWC should welcome be connected with the future one.

For the thesis topic, the SWC for the atomic straight drive can be implemented easily within the AUTOSAR framework by adapting to the characteristics of the application layer. In the next sub-section, the RTE layer of AUTOSAR will be discussed.

4.3.1.2 Runtime Environment (RTE) Layer

The RTE layer is the core of AUTOSAR architecture. It is designed to communicate between the hardware-independent layer, “Application,” and the hardware-dependent layer, “Basic Software.”[28] So, the main task of the RTE layer is to make the application layer independent from the specific ECU configuration in the lower layers. This characteristic of RTE allows it to provide the required communication and infrastructure services needed for system operations. Some main features of the RTE layer are provided below:

SWC Communication: RTE provides a real-time communication ability for the developed SWC within the architecture. As for the thesis, it ensures real-time communication between the atomic straight driving software component placed in the application layer and other system components, like the actuator, CAN communication, and ADAS sensors. There are two types of RTE communication modes: explicit and implicit.[29] For example, information from motor speed and obstacle detectors is directed through the RTE for proper response management.

Scheduling SWC: This part is major for scheduling tasks or events. Task scheduling can be done within RTE based on priority.[28] Task scheduling confirms that the operations are synchronized. For atomic straight driving, it is necessary to arrange the task scheduling based on the critical level.

Mapping Flexibility: The RTE ensures that SWCs are deployable to any ECU within the system defined during the configuration process and dynamically manages the communication approaches to maintain system integrity. This dynamic RTE facility ensures mapping flexibility.

Integration with Basic Software: The RTE controls the interaction between the application-level logic and the BSW level, ensuring that straight-driving functionality utilizes essential services like actuator control and communication protocols.

System Validation: During the software in loop test (SIL) or integration time, RTE confirms to validate the software component's compatibility with the system.

Overall, the Runtime Environment (RTE) is an important part of the AUTOSAR framework, connecting application-level software and hardware-specific details. The RTE will ensure that the atomic straight driving pattern SWC can be easily integrated within the system, communicate effectively with other components, and operate in a safety-critical automotive domain for the thesis.

4.3.1.3 Basic Software (BSW) Layer

The Basic Software (BSW) layer is the foundation of AUTOSAR architecture, which works closely with the hardware. This layer contains multiple sub-layers: the Microcontroller Abstraction Layer (MCAL), the ECU Abstraction Layer, Complex Drivers, and Services Layers. [10] This layer's software components (SWC) have already been designed and integrated into the TUCminiCar. In this section, an analysis will be conducted to get a clear idea of how basic software components function so that the application software component can work with this layer's components. In this sub-chapter, sub-sections from the BSW layer will be discussed in detail.

- **Microcontroller Abstraction Layer (MCAL)**

MCAL is the lowest sub-layer in the BSW layer.[10] This layer directly communicates with the related hardware within the microcontroller through the driver software available in MCAL. The MCAL consists of the following module groups, which are present in the picture mentioned below. Depending on the requirements, the blue-

marked groups are analyzed as installed within the TUC demonstrator TUCminiCar. Inside MCAL, four basic modules are

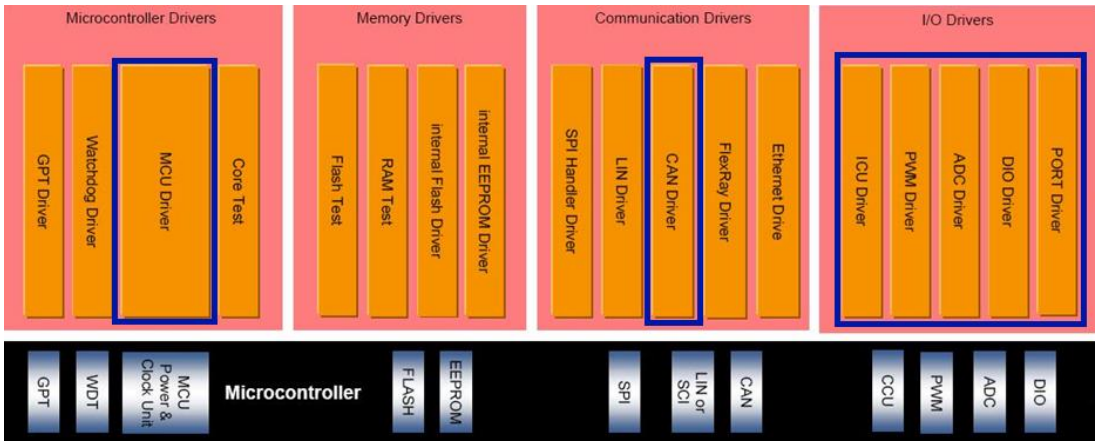

Figure 4.3: Microcontroller Abstraction Layer. [10]

presented: I/O Drivers, Communication Drivers, Memory Drivers, and Microcontroller Drivers. Two other drivers are also available: crypto drivers and wireless communication drivers. Since no modules are used in the demonstrator, they are not illustrated. No module is installed in the demonstrator for memory drivers, and memory specification is not required; rater memory management is done automatically by the ECU itself.

I/O Drivers: The input/output driver or I/O driver directly interacts with different types of hardware by separating the specifications of hardware types. This provides hardware independence to upper-layer modules. It also manages different signaling for sensors and actuators. Five I/O drivers are integrated into TUCminiCar; those are mentioned below:

i. *ADC Driver:* An Analog Digital Converter (ADC) driver converts all analog data to digital forms. This ADC driver works on the ADC channel group, which is analog pin inputs. [30]

ii. *DIO Drivers:* Digital Input Output (DIO) driver provides read and write access to the internal General-Purpose Input Output (GPIO) ports. The read-write operations are synchronous in general.[31]

iii. *PORT Driver:* The port driver defines the whole port structure of the microcontroller port pin. [31] This can be represented by configuring ports for reading sensor data or managing output data to control actuators.

iv. *PWM Driver:* The Pulse Width Modulation (PWM) driver generates PWM signals. It enables the duty cycle and signal period time to be selected.[32] The driver helps control actuators by varying the PWM signal, which provides different duty cycles.

v. *ICU Driver:* The input Capture Unit (ICU) driver is used to capture different events of the input unit, which does the Pulse Width Modulation (PWM) demodulation, tracking pulses of the signal, measuring frequency, and duty cycle.[32] This driver is required since the demonstrator uses a DC motor that generates PWM signals.

Communication Drivers: The communication drivers in the MCAL layer define communication services with microcontrollers. These drivers separate the upper layer as hardware independence since it lays into the lower layer and works closely with the hardware unit.[33] They support different communication protocols, such as Controller Area Network (CAN), Local Interconnect Network (LIN), FlexRay, Ethernet, etc. The CAN driver is used only for the thesis topic since the CAN hardware unit is used for communication. This CAN drive facilitates communication over CAN protocol by ensuring data transmission and receiving flow.

Microcontroller Drivers: Microcontroller drivers are a foundation for the microcontroller's hardware resources. They are used for internal peripherals with direct microcontroller access. [10] There are several key microcontroller drivers available in the AUTOSAR, which include the Microcontroller Unit (MCU), Watchdog Drivers, General Purpose Driver (GPT), etc. In the TUCminiCar, the Microcontroller Unit (MCU) driver is used. This driver provides clock and RAM initialization services.[34]

- **ECU Abstraction Layer**

In parallel with the MCAL layer's drivers, corresponding abstraction components

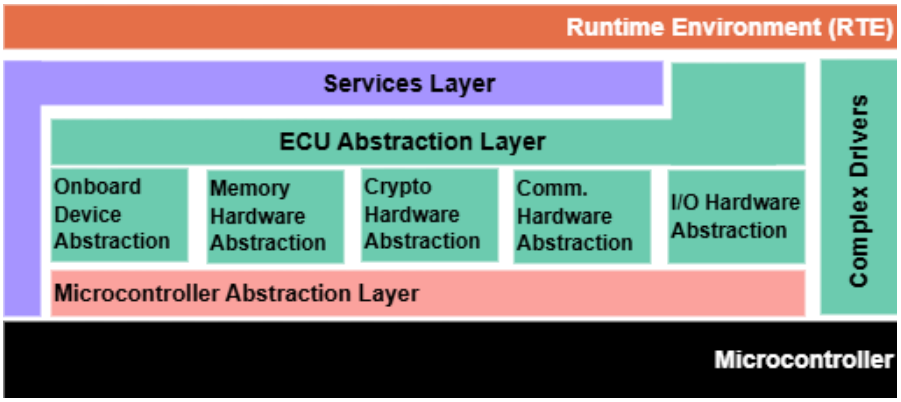

Figure 4.4: ECU Abstraction Layer's Module.

using the ECU Abstraction Layer is also available. This layer acts as an interface between low-level modules in MCAL and upper-layer modules in the Service layer, making the upper-layer hardware independent. The main components of this layer are I/O Hardware Abstraction, Communication Hardware Abstraction, Memory Hardware Abstraction, Crypto Hardware Abstraction, and Onboard Device Abstraction. [1]

- **Services Layers**

The services layer is the topmost layer in the BSW layer. It provides different services to the application layer's software components. The core components of the service layers are systems services, memory services, crypto services, off-board communication services, and communication services.[1] Some services provided by this layer are listed below-

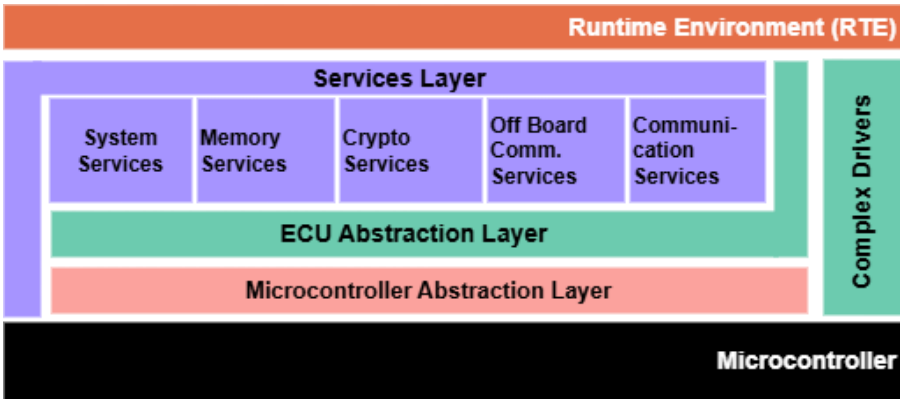

Figure 4.5: Services Layer's Module.

System Services: This feature of the service layer provides some required system functionalities, such as startup, shutdown, and mode management. The system handles state transitions, such as sleep or active modes.

Communication Services: It also provides a standardized interface for network communication, including protocol handling for different communication protocols like CAN, LIN, FlexRay, and Ethernet. This confirms continuous data exchange between Electronic Control Units (ECUs) and other networked devices. For the thesis topic, CAN communication is used. In the service layer, there is a protocol data router (PduR), which provides different PDUs (Protocol Data Units) connected with the low-level module CanIf (CAN Interface) in the ECU abstraction layer for communicating from upper to lower level.

Diagnostic Services: This service includes modules for onboard diagnostics (OBD) and fault detection and establishing communication with external diagnostic tools for maintenance and troubleshooting from a tester to ECU.

Memory Services: This service provides Application Programming Interfaces (APIs) for managing non-volatile memory (NVM), flash memory, and Electrically Erasable Programmable Read-Only Memory (EEPROM).

Safety and Security Services: Provide features such as watchdog monitoring, system health monitoring, and secure communication, which are related to functional safety standards like ISO 26262.

- **Complex Drivers**

Complex Device Driver (CDD) is the layer that can be accessed outside of AUTOSAR. This driver is designed so that those functionalities cannot be implemented by the basic software (BSW) components; complex drivers can do those. This driver can be accessed by AUTOSAR interfaces and/or the Basic Software module's API.[35]

For the thesis, it was necessary to analyze each module in the AUTOSAR architecture to design and integrate SWC for the atomic straight drive while maintaining the AUTOSAR standard. After analyzing the TUCminiCar, the developed AUTOSAR architecture overview is found as below picture:

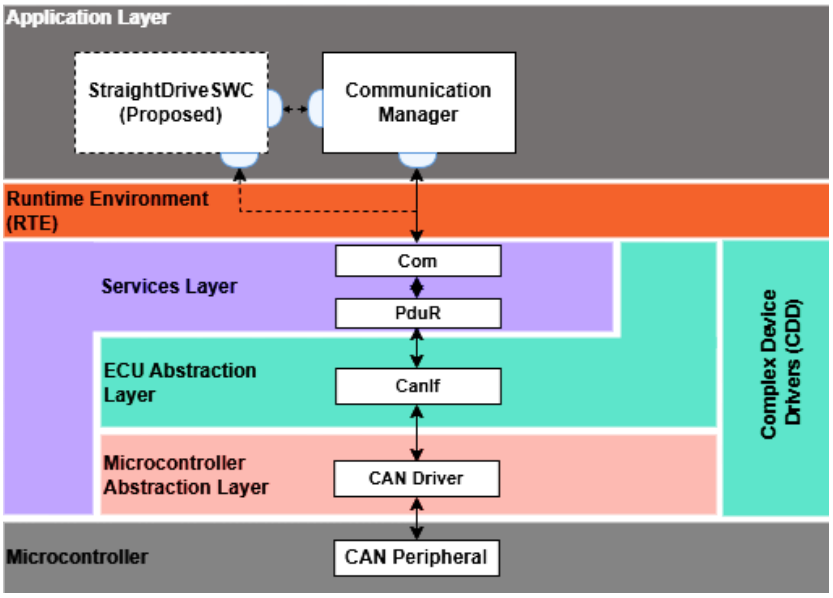

Figure 4.6: Overview of TUCminiCar AUTOSAR Architecture.

The application SWC StraightDriveSWC needs to be developed within the architecture shown in the proposal. For this purpose, the SWC template will be analyzed in the next sub-chapter to gather knowledge of the SWC creation process.

4.3.2 Software Component (SWC) Template

The software Component (SWC) template describes the procedure for developing SWCs within different layers of the AUTOSAR architecture.[36] SWCs can be different depending on the interaction, and their port type also varies depending on how they interact with other components. In this sub-chapter, the type of SWC in AUTOSAR and the core components related to SWC creation, like SWC Port Type, Interfaces, Internal behavior, and Runnable Entities, will be discussed.

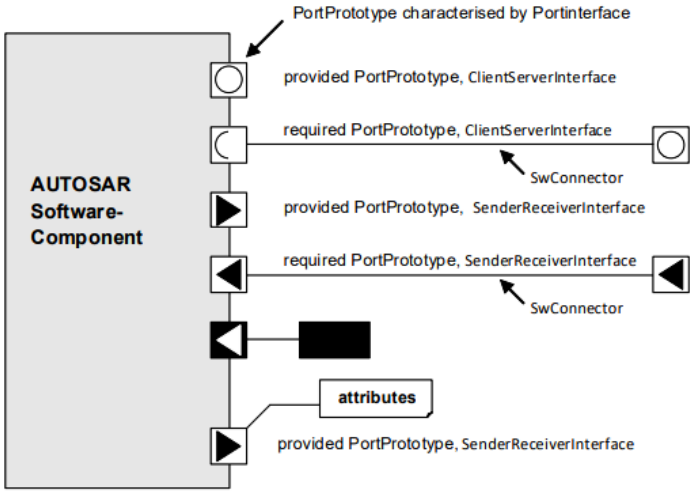

Figure 4.7: Graphical Representation of SWCs in AUTOSAR. [35]

- **Type of Software Component**

Software component creation depends on which layer and what responsibilities are assigned to each specific component. SWC needs to be designed based on the implementation requirement. The software components below are designed and implemented based on the thesis topic requirements.

ApplicationSwComponentType: These types of Software Components (SWCs) are hardware-independent and placed on the application layer of AUTOSAR architecture.[37] For the thesis topic, the required software component, “StraightDriveSWC,” which maintains atomic straight driving, needs to be an ApplicationSwComponentType software component. This software component can interact with different sensors and actuators via the “SensorActuatorSwComponentType” software component, which is in the BSW layer of AUTOSAR architecture. For sending data through the CAN bus, there is a requirement for another application type SWC, which has already been developed in the TUCminiCar and is named “CommunicationManager.”

SensorActuatorSwComponentType: These types of software components are fully hardware-dependent. A corresponding “SensorActuatorSwComponentType” Software

Component (SWC) is available for each hardware connected to the ECU. Although these types of SWCs are hardware-specific, they are still located above RTE and connect with ApplicationSwComponentType SWCs and EcuAbstractionSwC-omponentType SWCs and link them. For the TUCminiCar, several SensorActuator-SwComponentType SWCs have already been developed for different sensors and actuators; there is no requirement to create any new component for the thesis topic.[36], [38]

EcuAbstractionSwComponentType: These types of SWCs are typically designed to interact with Basic Software modules, specifically with the ECU Abstraction Layer. They are also hardware-specific SWCs, which contain references to specific hardware that interacts with different I/O ports of the ECU to access hardware.[36] Different EcuAbstractionSwComponentType SWCs have already been developed within TUCminiCar and, thus, do not need to be configured for the thesis topic.

CompositionSwComponentTypes: CompositionSwComponentTypes SWCs visualize the different developed SWCs for different layers. They also provide port-type information and help interconnect different SWCs.[35] These types of SWCs don't have any service-type ports but can visualize which internal ports are designed to connect externally. For the thesis topic, after designing the application SWC, CompositionSwComponentTypes SWC is required to connect with different SWC ports.[36], [39]

- **SWC Port and Interface Types**

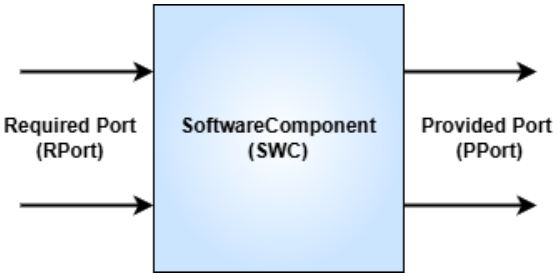

Figure 4.8: SWC Port Types.

Ports provide an interface for connecting different software components. They use connectors for connectivity. A port in a software component can be either a provided port (PPort) or a required port (RPort). A provided port is a type of port where the SWC sends data to another SWC, while required ports are used to receive data from another SWC. After creating ports in SWC, there will be a required interface within these ports for connectivity. These port interfaces can be of two types: sender-receiver and client-server. The sender-receiver interface is used when periodically data need to be sent and received, while the client-server is used when request-response data passing is required.[36], [40]

- **SWC Internal Behavior**

The internal behavior of an SWC defines its internal structure, which is built by combining runnable entities, implementation, RTE events, and data access definitions. An internal behavior should be created for each SWC. Within this internal behavior, there should be a runnable entity that needs to give data access to the created ports for the SWC. After that, it is required to define the triggering event for the runnable, such as the timing event, which is the most commonly used event in the automotive domain. After that, an implementation needs to be added to the internal behavior directory. This implementation will contain the “C” code file, where all the logic will be implemented for the designed SWC.[41][36]

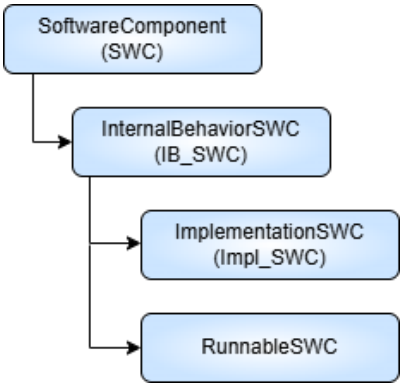

Figure 4.9: Internal Behavior Components.

Knowing the process guidelines is a prerequisite for any implementation. Since the application SWC will be required to control the straight drive for the ADAS demonstrator, it is necessary to know the standardized procedure for SWC development. This subchapter will help gather the necessary knowledge. In the next subchapter, the CAN communication method will be discussed.

4.4 Understanding Controller Area Network (CAN)

Controller Area Network or CAN is a serial communication protocol invented by the German company Bosch in the year 1986, and later, it was ISO standardized as ISO 11898 in the year 1993. CAN bus was introduced in the automotive domain due to the complexity of wiring within a car due to the increased ECU numbers. A standard “CAN” can have a data rate of 1 MBit/s data transfer rate. Two-wired high and low connectivity methods replaced all the traditional wires. In each ECU, there is a CAN transceiver that can send and receive data. CAN bus uses the CAN arbitration method to send or receive data, depending on the priority of the messages.[42], [43]

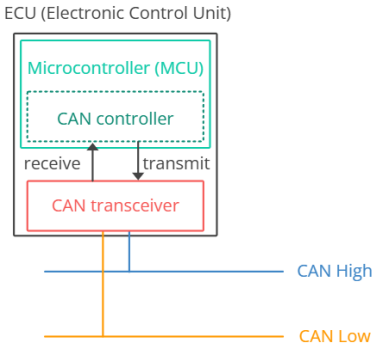

Figure 4.10: CAN Bus.[42]

4.4.1 Overview of the CAN Protocol

The Controller Area Network (CAN) protocol works based on carrier sense multiple access protocol with collision detection (CSMA/CD) protocol. Here, every node

connected to CAN bus communication needs to check the bus load status and then send a message. The message arbitration happens based on the priority of the message; thus, CAN bus communication is reliable in the automotive domain. Depending on the identifier, the Controller Area Network (CAN) has been standardized into two types: Standard Frame Format (CAN 2.0A) and Extended Frame Format (CAN2.0B).[44], [45]

- **Standard Frame Format**

The identifier's length for the Standard Frame is 11 bits. The format of the standard frame is mentioned in the below picture, and different terms in the picture are described afterward.[44], [46]

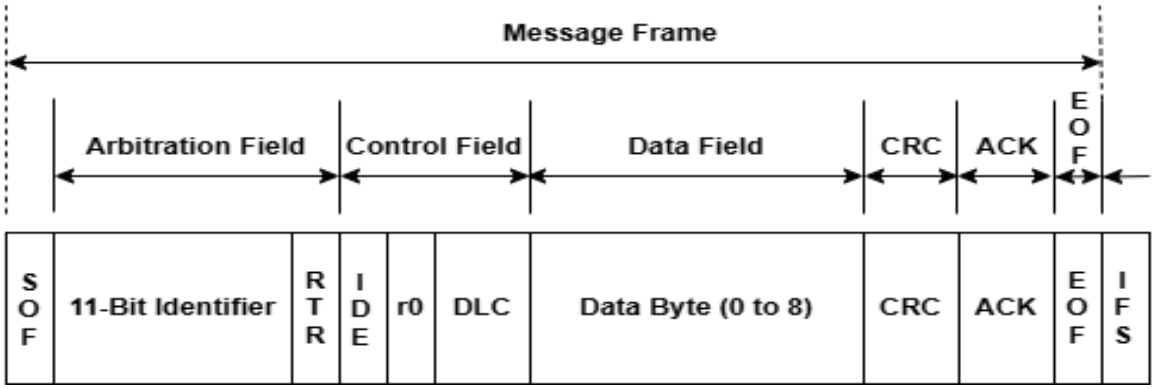

Figure 4.11: CAN (2.0A) Standard Frame Format.[43][45]

SOF: Start of Frame (SOF) defines the starting of the data frame, which contains a single bit that needs to be dominant.

Identifier: The Identifier length for the standard frame is 11 bits; this is the arbitration field that defines the priority.

RTR: Remote Transmission Request (RTR), which differentiates between data frame and remote frame. Dominant (0) is for the data frame, and Recessive (1) is for the remote frame.

IDE: Identifier Extension (IDE) bit defines whether the standard or extended frame format.

r0: Reserved bit for future uses, need to be dominant (0).

DLC: The Data Length Code (DLC) defines the number of bytes in the data field. DLC is 4 bits wide, as shown in the table below. Where ‘d’ stands for dominant and ‘r’ is for recessive.

Table 4.1: DLC Define Number of Data Bytes.[43]

Number of Data Bytes	Data Length Code (DLC)			
	DLC3	DLC2	DLC1	DLC0
0	d	d	d	d
1	d	d	d	r
2	d	d	r	d
3	d	d	r	r
4	d	r	d	d
5	d	r	d	r
6	d	r	r	d
7	d	r	r	r
8	r	d	d	d

Data Field: This field indicates that the data is being transferred. The number of data bytes indicates the payload (data byte) length, which can range from 0 to 8 bytes; others are not accepted.

CRC: The cyclic redundancy Check (CRC) detects errors in the data field (15 bits of CRC+1-bit delimiter).

ACK: The Acknowledgement (ACK) field contains 2 bits for providing acknowledgment status, where the first bit is for acknowledgment status and the second one is a delimiter.

EOF: End of Frame (EOF) containing seven recessive bits provides end status.

IFS: Interframe Space (IFS) contains 3 bits of an idle period, providing a mandatory delay between consecutive frames.

- **Extended Frame Format**

The main difference between the extended CAN (2.0 B) frame format and the standard one is the extension of the identifier field. In the extended format, another 18 bits are added for the identifier, for a total of 29 bits. The structure of the extended frame format is as per the below figure, and the modification from the standard frame format follows afterward.

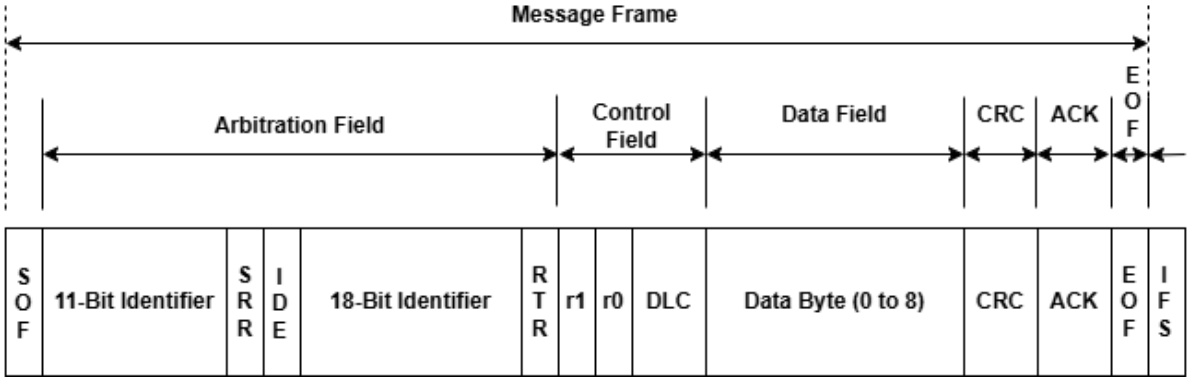

Figure 4.12: CAN (2.0B) Extended Frame Format.[43][45]

Bit Identifier: 29-bit identifiers in total, which replaced the 11 bits from the standard one.

IDE: IDE = 0, standard frame format (11-bit identifier), IDE = 1: Extended Frame Format (29-bit identifier)

SRR: Substitute Remote Request (SRS) is added in the extended frame and is always recessive to ensure the standard frame gets higher priority over the extended one for the same base identifier (11).

All the other fields remain the same in the Extended Frame as in the Standard frame.

4.4.2 Overview of CAN in TUCminiCar

After analyzing "TUCminiCar," it is clear that the CAN communication stack modules have already been integrated within the car for communication purposes. The integrated modules are shown in the abstract picture below, and details are mentioned afterward.

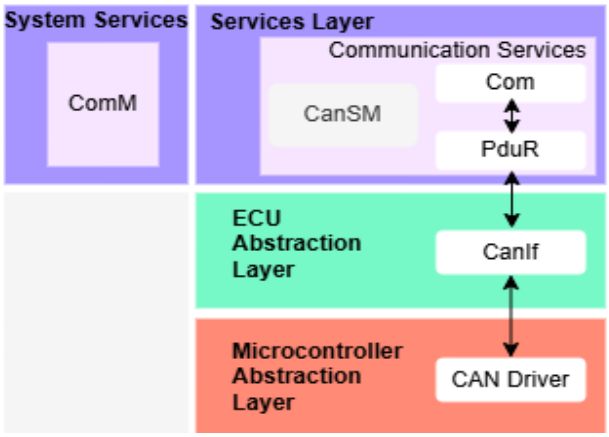

Figure 4.13: TUCminiCar CAN Stack Overview.

CAN Driver: This module belongs to the Microcontroller Abstraction layer and interacts directly with the CAN controller hardware. Standard CANIdType is configured for the TUCminiCar.

CanIf: The CAN Interface (CanIf) module is placed in the ECU abstraction layer and acts as an interface between the upper- and lower-layer modules.[47] Transmission and receiving channels are created in CanIf modules for the demonstrator.

PduR: The Protocol Data Unit Router (PduR) connects all the PDUs created in the system by routing them in a routing table. This router is placed in the communication service, which is in the service layer.[48] Different PDUs are created in the demonstrator to send data for controlling the car, sending distance sensor data, and knowing the car's status.

Com: The Communication (Com) module is placed in the communication service layer, connecting PduR to RTE and staying between them. This module is used for signaling purposes by using Interaction Layer Protocol Data Unit (I-PDU).[49] For the TUCminiCar, the number of PDUs is created, and corresponding I-PDUs are also created in the Com module for signaling purposes.

CanSM: The CAN State Manager (CanSM) is a module in the communication service layer that manages the state of the CAN network.[50] For the demonstrator, a single CAN network has been created, and CarEcu is assigned to that network to control the state.

ComM: The Communication Manager (ComM) module is placed in the system services layer and manages the ECU's communication states.[51] For the demonstrator, a single network channel has been added to this module.

4.5 Analyze “TUCminiCar” System Configuration

This subchapter provides the high- and low-level existing configuration overview for the “TUCminiCar,” which will be used as the base for implementing the project. The system configuration can be viewed as follows-

- **dSPACE SystemDesk Configuration Overview**

After analyzing the current configuration from TUCminiCar, it is found that Basic Software system designing is already visible in the CompositionSWC of dSPACE SystemDesk, which is extracted as a high-level overview as below:

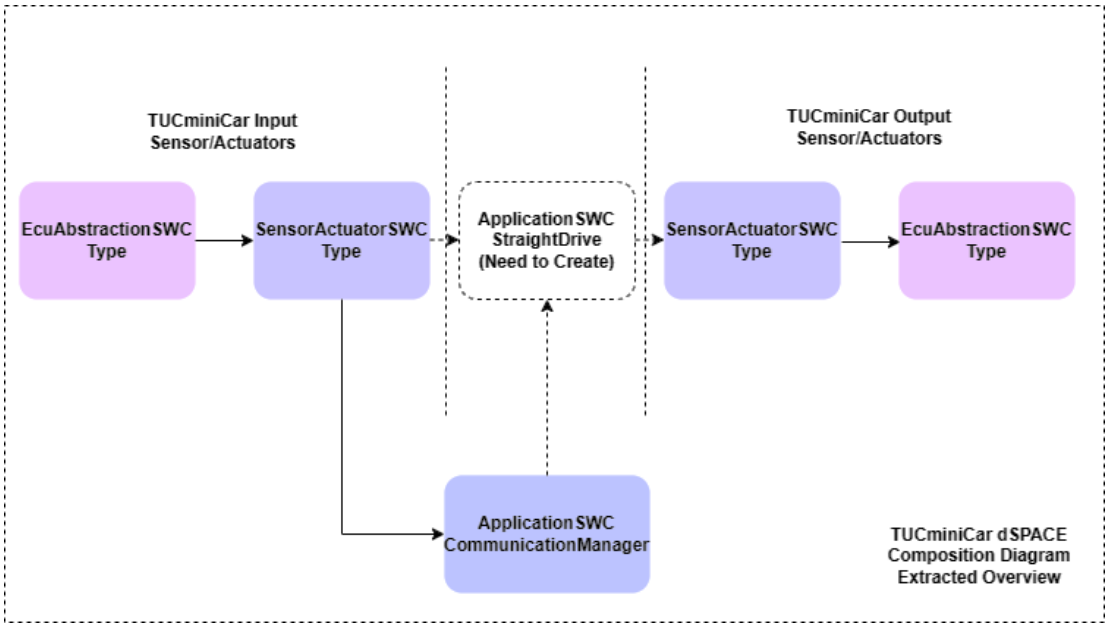

Figure 4.14: TUCminiCar dSPACE Composition Overview.

In the above composition overview, only the StraightDrive application SWC is not present; others are extracted from developed SWCs. For interacting with different hardware units, different EcuAbstractionSw-ComponetType SWC have already been designed and integrated within the dSPACE SystemDesk file *.sdp. For the thesis topic, there is no requirement to go further for the connectivity analysis to control the sensor or actuators. Since the target Application Software Component will interact with SensorActuatorSWCType to control the demonstrator, it is necessary to analyze their interfaces and working logic to deploy the StraightDriveSWC. For analyzing all the required SWCs and ports, the data from the components are arranged in below the three tables: one is for the input side, another for communication, and the last one is from the output side, as below-

- **TUCminiCar Input Sensor/Actuator SWC**

The table below provides all the required input sensor/actuator data (speed, steering angle, and obstacle data) sources from existing SWCs to implement and make connectivity for the application SWC StraightDrive development.

Table 4.2: TUCminiCar Input Parameter.

SWC Type	SWC Name	Port Name	Port Type	Interface	Purpose
SensorActuatorSwComponentType	SensActInputSonar	pp_inSonarValues	Provided Port	sri_sonar	This port from SWC provides obstacle data from the Sonar sensor.
	SensActInputEncoder	pp_inSpeedValue	Provided Port	sri_float	This port from SWC provides the current speed from the DC motor.
	SensActInputAnalog	pp_inSteeringPot	Provided Port	sri_float	This port from SWC provides the current steering angle from the servo motor.

- **TUCminiCar Communication SWC**

The table below is for the communication manager SWC. Apart from all other ports for internal communication, this SWC has eight user input ports available to send data from the tester's end. All the ports have the same port type and interface; users can use these ports to send the required data to the ECU to control the TUCminiCar after receiving and processing it within the planned application SWC StraightDrive.

Table 4.3: TUCminiCar CommunicationManager Parameter.

SWC Type	SWC Name	Port Name	Port Type	Interface	Purpose
ApplicationSwComponentType	CommunicationManager	pp_CanRx_UserControl1	Provided Port	sri_uint8	Send data from the tester to ECU through CAN Bus
		pp_CanRx_UserControl2	Provided Port	sri_uint8	Send data from the tester to ECU through CAN Bus
		…..	Provided Port	sri_uint8	Send data from the tester to ECU through CAN Bus
		pp_CanRx_UserControl8	Provided Port	sri_uint8	Send data from the tester to ECU through CAN Bus

- **TUCminiCar Output Sensor/Actuator SWC**

Below is the parameter table for the output side of TUCminicar's configuration parameter in dSPACE. From all the configured ports in the system, only those required for the thesis topic are analyzed in the table below, which will be used to connect in the application SWC StraightDrive to control different actuators of the car.

Table 4.4: TUCminiCar Output Parameter.

SWC Type	SWC Name	Port Name	Port Type	Interface	Purpose
SensorActuatorSwComponentType	SensActOutputChassis	rp_outMotorDuty	Required Port	sri_uint8	This port receives moto duty value to accelerate the car
		rp_outSteeringDuty	Required Port	sri_uint8	This port receives steering duty value to fix the steering angle
	SensActOutputBeeper	rp_outBeeperMode	Required Port	sri_uint8	This port receives instructions for beeper control
	SensActOutputLights	rp_outHighBeam	Required Port	sri_bool	This port receives instructions for HighBeam control
		rp_outLowBeam	Required Port	sri_bool	This port receives instructions for LowBeam control
		rp_outLeftSignal	Required Port	sri_bool	This port receives instructions for LeftSignal Light Control
		rp_outRightSignal	Required Port	sri_bool	This port receives instructions for RightSignal Light Control
		rp_outReverseLights	Required Port	sri_bool	This port receives instructions for Reverse Light Control
		rp_outBrakeLights	Required Port	sri_bool	This port receives instructions for Brake Light Control

After analyzing the above parameter information, a clear idea was obtained for developing the SWC application and its connectivity with existing SWCs in the demonstrator. After developing the application SWC, it needs to be validated and a new *.arxml file needs to be generated in SystemDesk. This file can then be integrated into EB Tresos studio. The implementation, testing, and integration steps will be described in the Implementation chapter.

5 Implementation

This chapter presents the practical application of the thesis topic “AUTOSAR Software Component for Atomic Straight Drive Patterns” to the TUCminiCar demonstrator. The knowledge and information gathered throughout the paper before this chapter have formed a base for the project implementation. The following figure illustrates the high-level overview of the project implementation.

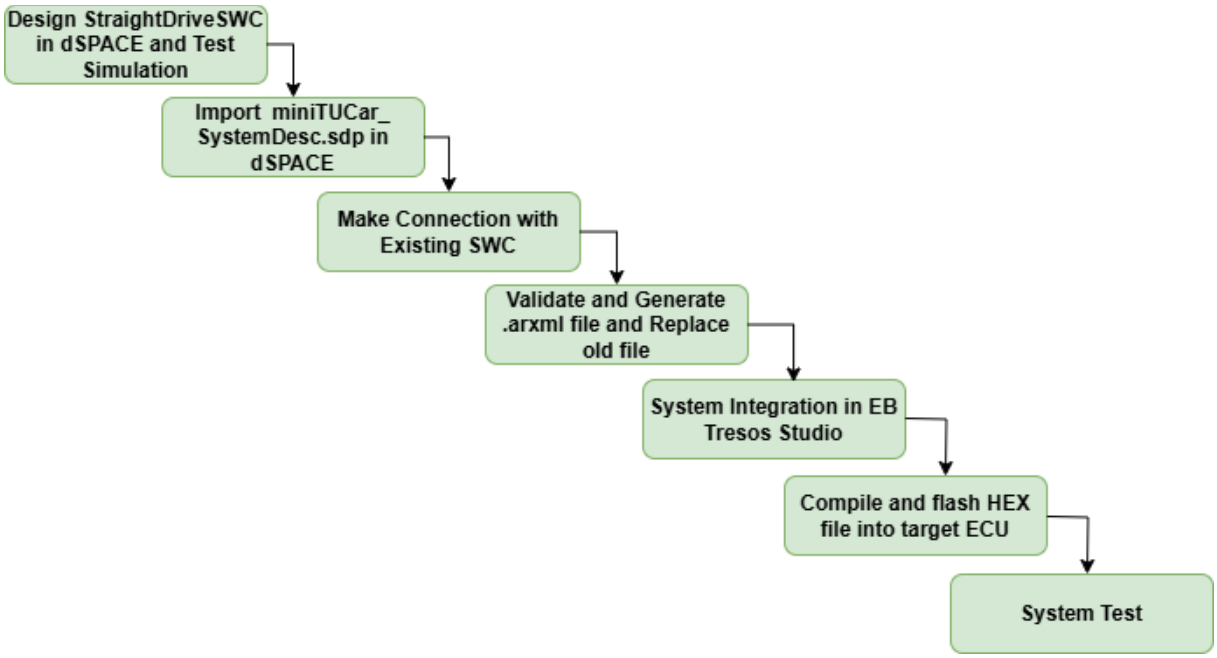

Figure 5.1: Implementation Overview.

Based on the project implementation, this chapter has been divided into four sub-chapters: SWC Development (5.1), Software-In-Loop (5.2), System Integration (5.3), and System Test (5.4). These sub-chapters will be illustrated in detail throughout this chapter.

5.1 SWC Development

This chapter will focus on developing the application software component for the atomic straight driving pattern. dSPACE SystemDesk will be used for design and logical implementation. The SystemDesk 5.6 version has been used for the implementation at the dSPACE end. In this sub-chapter, the following sections will be illustrated: Application SWC Designing (5.1.1), Interface Definition (5.1.2), Logic Implementation (5.1.3), and RTE Generation (5.1.4).

5.1.1 Application SWC Designing

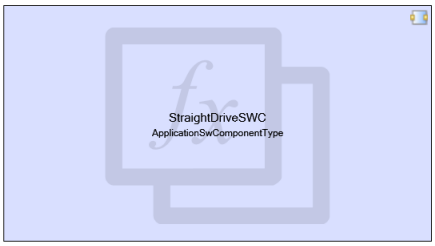

Figure 5.2: StraightDriveSWC Creation.

This section will describe the design of the application software component “StraightDriveSWC” in dSPACE. First, the existing System Description file “miniTUCar_SystemDesc.sdp” needs to be imported into dSPACE. From the existing configuration, there is a dedicated place for application SWC design; the “StraightDriveSWC” application-type software component needs to be created in that section, which is named the “ApplicationSwComponents” folder.

This SWC component will handle different elements like engine control, speed control, and steering control, receive obstacle data from a sonar sensor, and also handle various light and auditory components. Depending on the compatibility of the existing basic software (BSW) component and its operation, it is necessary to create the required and provided port, as discussed already in chapters (4.3.2) and (4.5).

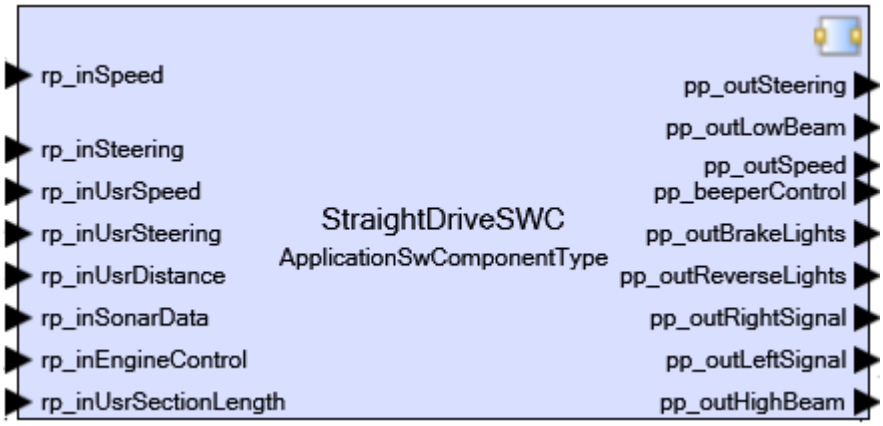

Figure 5.3: StraightDriveSWC Port Definition

Depending on the requirement, the provided port and required port have been created within the SWC as per the above picture. The required ports are initiated with “rp_,” whereas the provided ports are “pp_.” All the required ports (rp) are to receive data from other software components, and SWC will take the necessary steps to process the received data; after that, those will be transferred to the appropriate areas through the provided ports (pp). In the next sub-chapter, interfaces will be assigned to the ports.

5.1.2 Interface Definition

This subchapter describes how the application software component interacts with the other SWCs. It defines how communications are managed within different SWCs. The interface assigning depends on how each port will interact with others. The connected

ports should have a similar interface. Otherwise, there will be compatibility issues and no communication between them. As knowledge gathered from subchapter (4.3.2) about SWC interface type and (4.5) about existing interface types, the port assignment has been executed as per the below figure.

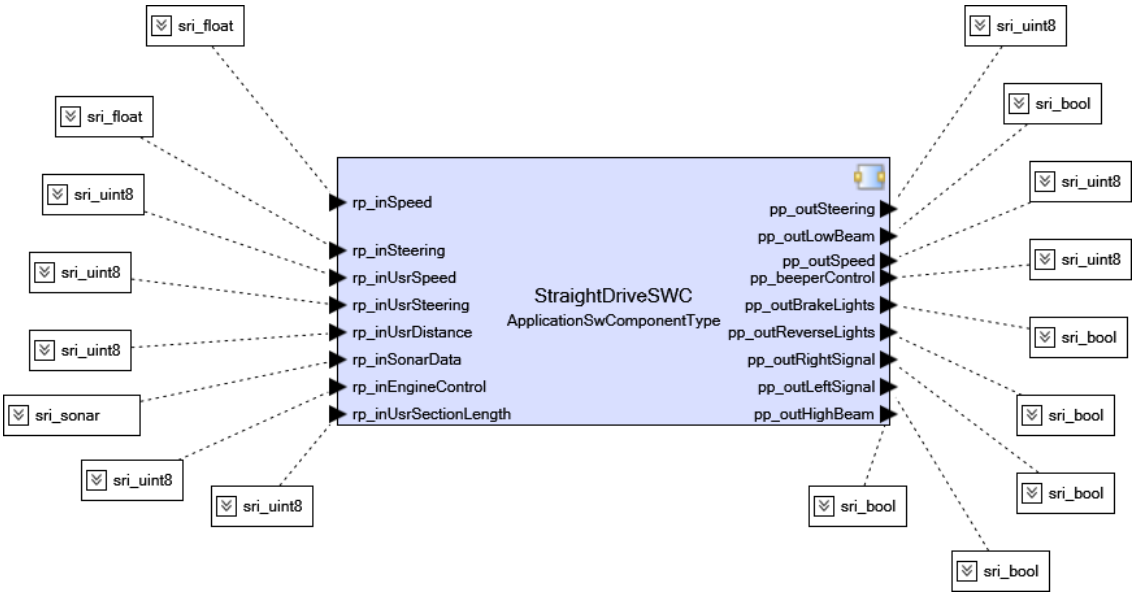

Figure 5.4: StraightDriveSWC Interface Definition.

All the interfaces are sender-receiver in type, and the interfaces assigned to each port that already existed in the SystemDesc file have only been assigned to the ports depending on the requirement. Ports and assigned interfaces can be categorized into two sections: Provided Port (PPort) Interfaces and Required Port (RPort) Interfaces. For each section, the port, interface, and purpose of the ports are arranged in the tables below.

Required Port Interfaces: The table below lists all the Required Ports and assigned interfaces along with purposes that have been configured for “StraightDriveSWC.”

Table 5.1: StraightDriveSWC RPort with Interface Definition.

Port Name	Port Type	Interface	Purpose
rp_inEngineControl	RPort	sri_uint8	Receive Engine Control from user/tester
rp_inUsrSpeed	RPort	sri_uint8	Receive Speed data from the user/tester
rp_inSpeed	RPort	sri_float	Receive Speed data from Car
rp_inUsrSteering	RPort	sri_uint8	Receive Steering data from the user/tester
rp_inSteering	RPort	sri_float	Receive Steering data from Car
rp_inSonarData	RPort	sri_sonar	Receive object detection data from sonar sensor
rp_inUsrDistance	RPort	sri_uint8	Receive path section data from the user/tester, like how many sections need to travel
rp_inUsrSectionLength	RPort	sri_uint8	Receive each Section Length data from the user/tester

Provided Port Interfaces: The table below lists all the Provided Ports and associated interfaces along with purposes that have been configured for “StraightDriveSWC.”

Table 5.2: StraightDriveSWC PPort with Interface Definition.

Port Name	Port Type	Interface	Purpose
pp_outSteering	PPort	sri_uint8	Provide data to the car for Steering control
pp_outSpeed	PPort	sri_uint8	Provide data to the car for Speed control
pp_outLowBeam	PPort	sri_bool	Provide data to the car for low-beam control
pp_outHighBeam	PPort	sri_bool	Provide data to the car for high-beam control
pp_outLeftSignal	PPort	sri_bool	Provide data to the car for Left Signal Light control
pp_outRightSignal	PPort	sri_bool	Provide data to the car for the Right Signal Light control
pp_outBrakeLights	PPort	sri_bool	Provide data to the car for Brake Light control
pp_outReverseLights	PPort	sri_bool	Provide data to the car for Reverse Light control
pp_beeperControl	PPort	sri_uint8	Provide data to the car for Beeper control

5.1.3 Logic Implementation

The StraightDriveSWC controls different driving scenarios, such as distance tracking, obstacle detection, speed control, and steering control. It reads various sensor inputs and user input data and, after that, processes the appropriate computations and sends control signals to the actuators in the vehicle. After assigning ports and interfaces to the SWC, it is time to implement logic so that these ports can interact with proper data control. The implementation logic for each key point is discussed below-

Engine Control Logic Flow: SWC reads the engine control status from the user input and then processes it as follows-

- Condition 1: Engine OFF

“If rp_inEngineControl == 0”, the vehicle is placed in a neutral state:
Speed and steering values are set to the default (neutral) state.
No further processing of inputs is allowed.

- Condition 2: Engine ON

“If rp_inEngineControl == 1”, the engine is on, and the car is ready to take new user inputs, such as speed, steering, and distance.

- Condition 3: Undefined Values

For all other values of “rp_inEngineControl,” no further operations are performed.

Distance Calculation Logic Flow: SWC can take speed data from two sources: user input through a CAN message or from inside the car. To accelerate the car, user input data “rp_inUsrSpeed” is considered, and for calculating current speed from inside the car, “rp_inSpeed” is considered for the actual value consideration. The car receives target distance (in millimeters) data from the multiplication value of “rp_inUsrDistance” and “rp_inUsrSectionLength” (rp_inUsrSectionLength, each atomic section, 1=200mm, 2 = 400mm, 3 = 600mm, and so on…) from a tester.

- Step 1: System Initialization
  Initialize the system and reset speed, cumulative distance, and elapsed time to zero.

- Step 2: Engine Validation
  Check the engine status and be ready to accept user input if the engine is on.

- Step 3: Input Data Processing
  Read user and car input data for speed, steering, and target distance and keep updating in real-time.

- Step 4: Distance Tracking
  The SWC calculates the traveled distance based on the current speed (mm/s) and elapsed time. To track elapsed time, the control loop is executed every 10 ms, and distance calculations continue until the target distance is reached.
  Distance Calculation: distance Increment = speed (mm/s) × time Interval (s);
  required iterations = target distance (mm) ÷ distance increment (mm);
  cumulative distance = distance increment (mm) × no. of iterations.

- Step 5 (Control Output): Based on the cumulative distance tracking, once the distance is reached, write neutral duty cycle (100) to the “pp_outSpeed” port to activate the brake; before that, keep writing forward (>100) or revere (<100) duty cycle as per the user input provided. To maintain a straight drive path, keep writing steering duty cycle neutral (100) to output port pp_outSteering.

Obstacle Detection Logic Flow: SWC will receive obstacle detection data at the port “rp_inSonarData.” Based on the car's current condition, the brakes will be activated to stop.

- Step 1: Read Sonar Data
  Read sensor data at the port “rp_inSonarData,” which is the array of sonar sensors on the front [0,1,2] and back [8,9] sides of the car.

- Step 2: Obstacle Detection
  Detect any object within 200 mm of the sensors.

- Step 3: Braking Mechanism
  If the car is running, brake it immediately with neutral duty (100) value, and stop the distance tracking counter to stop measuring cumulative distance.

- Step 4: Obstacle Elimination
  If the obstacle is removed, the car keeps moving automatically until the target distance is reached.

Signaling Logic Flow: Based on the processed data from the SWC, in response to the car's operation, some visual and auditory actuators within the car also get activated or deactivated throughout the ports defined in the SWC.

- Beeper Control
  Whenever the sonar sensor detects an obstacle, the car will continue to beep until the obstacle is removed from the detectable distance.

- Brake Lights Control
  When the brake is activated, either during obstacle detection or traveling the target distance, the brake lights on the back side of the car turn on. In normal conditions, these are turned off.

- Low Beam Control
  When the car moves forward direction under the duty cycle of 150 (equivalent to 630 mm/s), the Low Beam lights are on. For the neutral and reverse directions, these are off.

- High Beam Control
  High beam lights are on when the forward speed is equal to or greater than that of duty cycle 150. In other conditions, these are off.

- Reverse Lights Control
  These lights are on whenever the car moves in the reverse direction; in other conditions, they are off.

- Left Signal Control
  When the steering turns in the left direction (duty cycle value 101 to 200), the left signal light starts blinking and is off in any other situation (0 to 99).

- Right Signal Control
  The right signal control light starts blinking when the steering turns to the right side (0 to 99 duty cycle); in other conditions, the light remains off.

To implement the logic mentioned above, “Internal Behaviour” within the SWC needs to be created, where the implementation of the “C” code will be placed.

5.1.4 SWC Internal Behavior

The internal behavior mainly describes Runnables and the Implementation of SWC. For this purpose, “IB_StraightDriveSWC” internal behavior has been created within the application SWC. Inside internal behavior, implementation “Impl_StraightDriveSWC” and runnable “StraightDriveSWC” were also created.

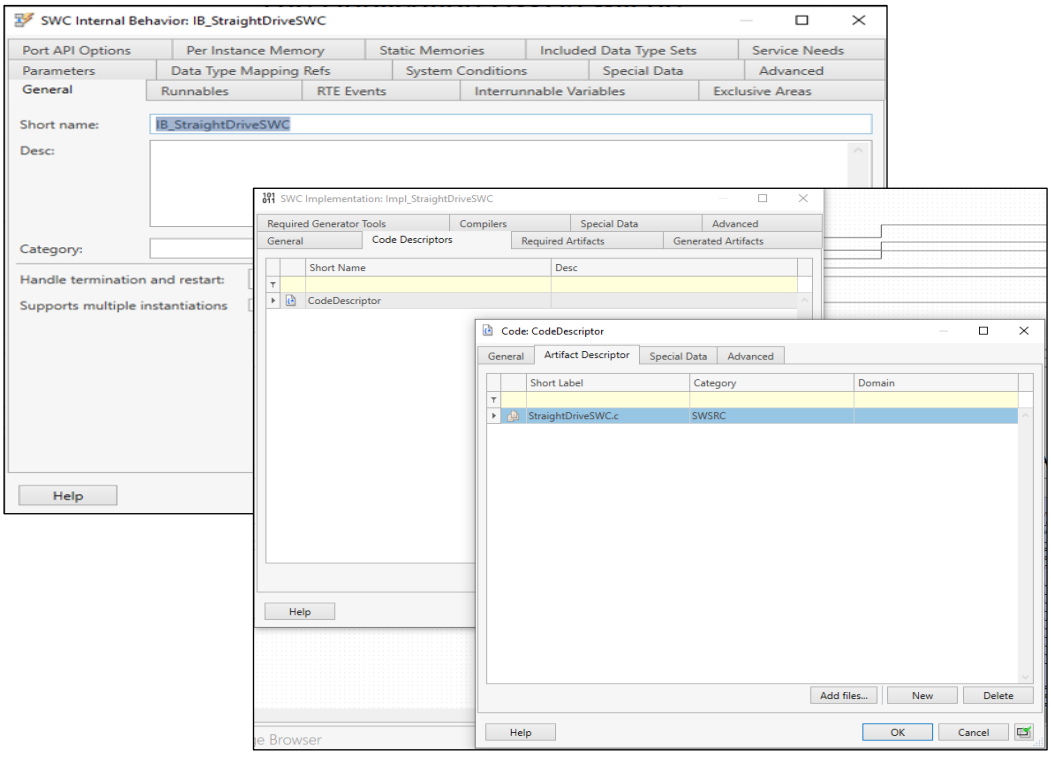

Figure 5.5: StraightDriveSWC CodeDescriptor.

Inside the “Impl_StraightDriveSWC,” memory resource consumption needs to be assigned for the “C” file with the logic implemented, named “StraightDriveSWC.c.” Then, a new “CodeDescriptor” needs to be added, and inside this, the *.c file needs to be added. In the next subsection, RTE Generation, along with the runnable entity, will be described.

5.1.5 RTE Generation

The Runnable “StraightDriveSWC” created inside the internal behavior needs to be assigned a triggering event. This event will trigger the runnable by RTE as per the mentioned method. For the SWC, the RTE event is defined as a “Timing Event,” and

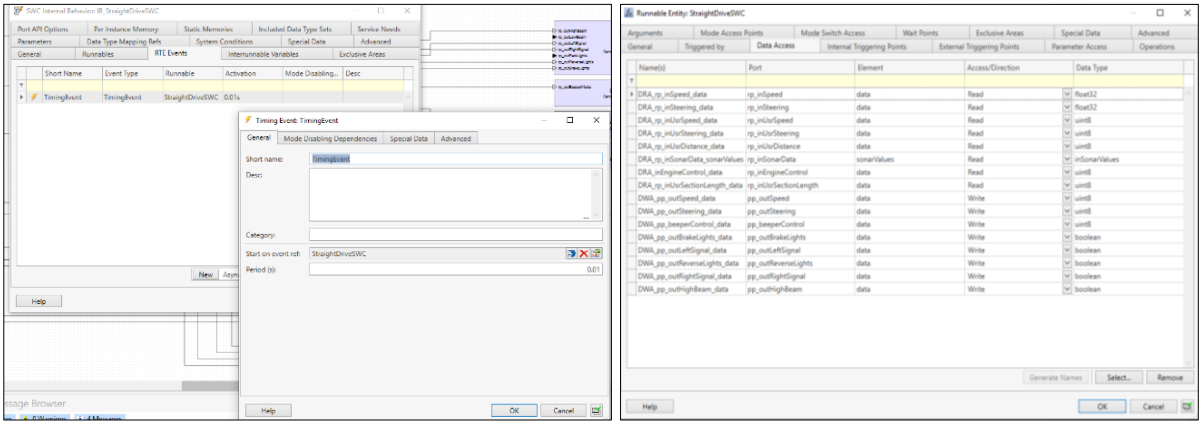

Figure 5.6: StraightDriveSWC RTE Event and Data Access.

it will be triggered every 10ms. After that, all the ports created within the SWC are provided data access for the Runnable Entity. Now, these ports are ready to read and write data from or to the desired SWC ports. The RPorts will read data as per the below RTE API functions:

```
/* Read inputs */
uint8 engineControlStatus = Rte_IRead_StraightDriveSWC_rp_inEngineControl_data();
uint8 userSpeedValue = Rte_IRead_StraightDriveSWC_rp_inUsrSpeed_data();
uint8 userSteeringValue = Rte_IRead_StraightDriveSWC_rp_inUsrSteering_data();
float32 globalCarSpeedValue = Rte_IRead_StraightDriveSWC_rp_inSpeed_data();
float32 globalCarSteeringValue = Rte_IRead_StraightDriveSWC_rp_inSteering_data();
uint16* sonarData = Rte_IRead_StraightDriveSWC_rp_inSonarData_sonarValues();
uint8 targetDistanceSections = Rte_IRead_StraightDriveSWC_rp_inUsrDistance_data();
uint8 usrSectionLength = Rte_IRead_StraightDriveSWC_rp_inUsrSectionLength_data();
```

After SWC processes the input data, PPorts will write data according to the following Runtime Environment (RTE) API functions:

```
/* Write Outputs */
Rte_IWrite_StraightDriveSWC_pp_outSpeed_data(motorDutyValue);
Rte_IWrite_StraightDriveSWC_pp_outSteering_data(steeringDutyValue);
```

```
Rte_IWrite_StraightDriveSWC_pp_beeperControl_data();
Rte_IWrite_StraightDriveSWC_pp_outLeftSignal_data();
Rte_IWrite_StraightDriveSWC_pp_outRightSignal_data();
Rte_IWrite_StraightDriveSWC_pp_outReverseLights_data();
Rte_IWrite_StraightDriveSWC_pp_outHighBeam_data();
Rte_IWrite_StraightDriveSWC_pp_outLowBeam_data();
Rte_IWrite_StraightDriveSWC_pp_outBrakeLights_data();
```

Now, all the ports are ready to be connected with respective ports in other SWCs named SensorActuatorSWCType and CommunicationManager. After connecting to the desired ports of those SWCs, the composition diagram extract will be like as below picture:

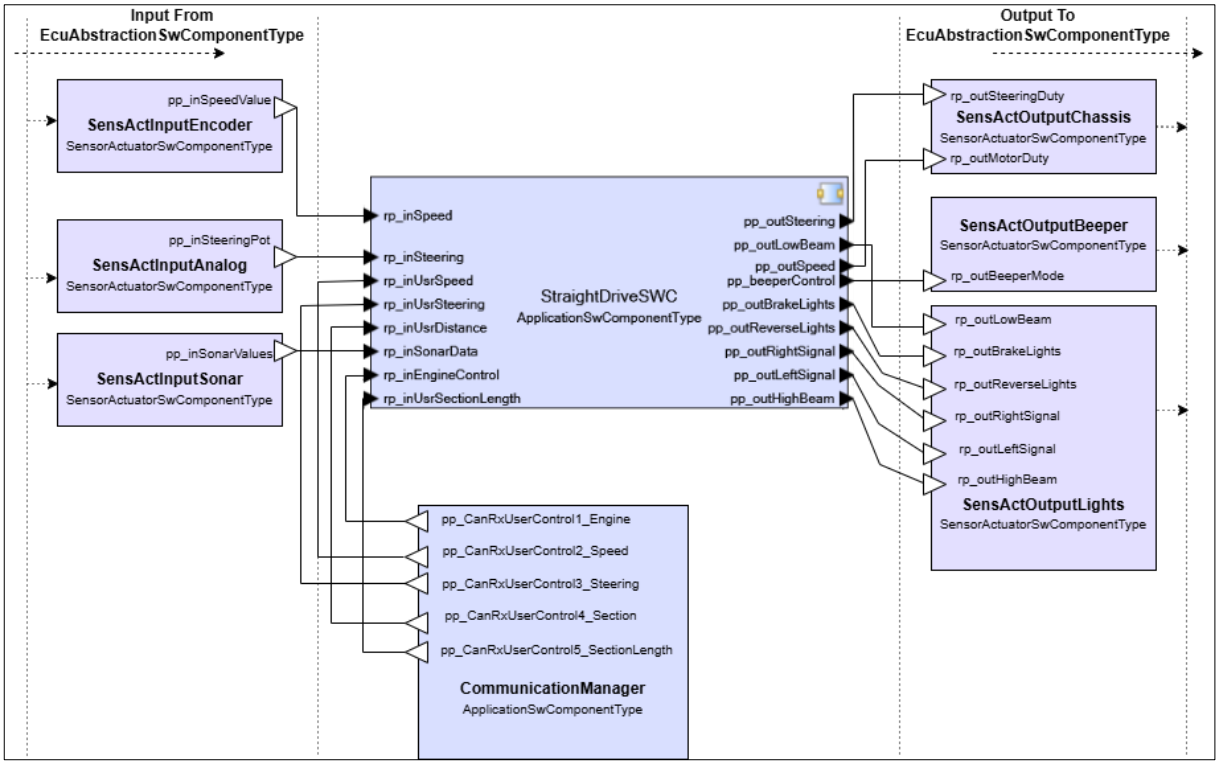

Figure 5.7: TUCminiCar Composition Diagram Extract.

Now, it is necessary to do the SWC-to-ECU mapping for the newly added application SWC, and thus, need to select “CarEcu” for this mapping. Finally, check the project validation, which will ensure the design procedure after a successful validation. After successful validation, the SystemDesc file has been updated and is ready to generate a new *.arxml file to replace the old one.

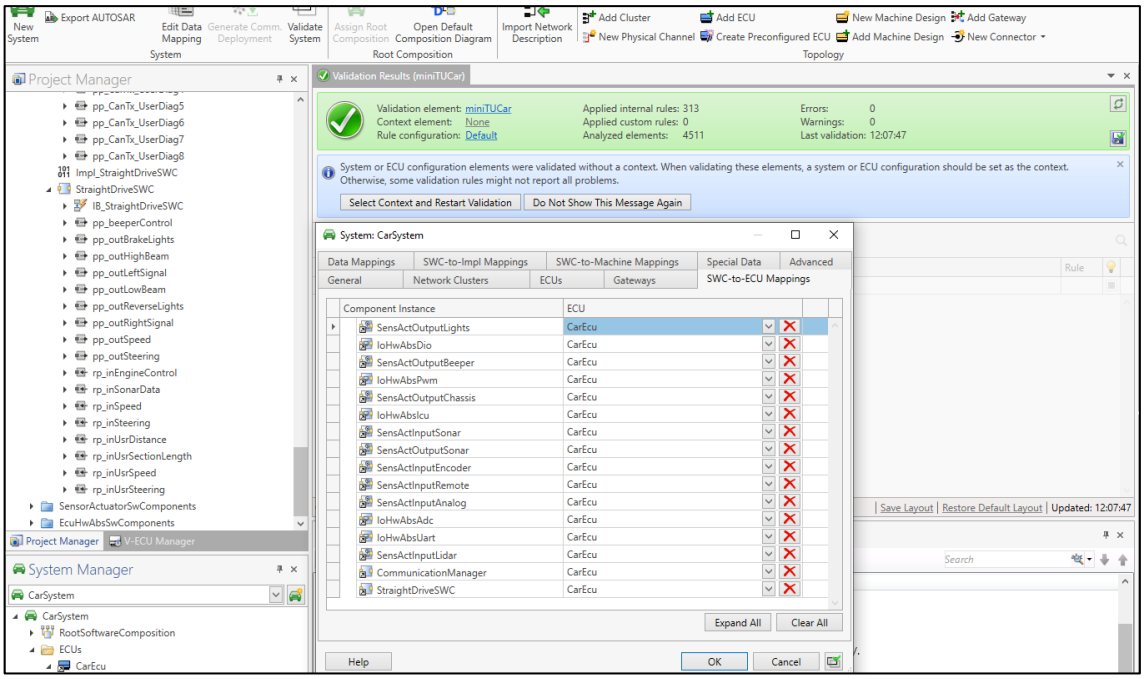

Figure 5.8: CarEcu Mapping and Validation.

The functionality of the developed SWC needs to be checked before integration into the EB Tresos studio. To do this, a Software-In-Loop test is required, which will be discussed in the next sub-chapter.

5.2 Software-In-Loop (SIL) Testing

Software-In-Loop (SIL) testing is the most commonly used method in the automotive industry to check SWCs' functionality. In this sub-chapter, a developed SWC will be tested in a simulation environment before integration.

5.2.1 StraightDriveSWC Prototype

Since the SWC was developed under the whole system architecture for the TUCminiCar, it is necessary to separate it and test it in a simulation environment only to test its functionality. To do so, a prototype, “StraightDriveSWC,” has been developed, which is similar to the originally developed component. Only basic driving functionality, along with obstacle detection, will be tested, as these are the main safety-

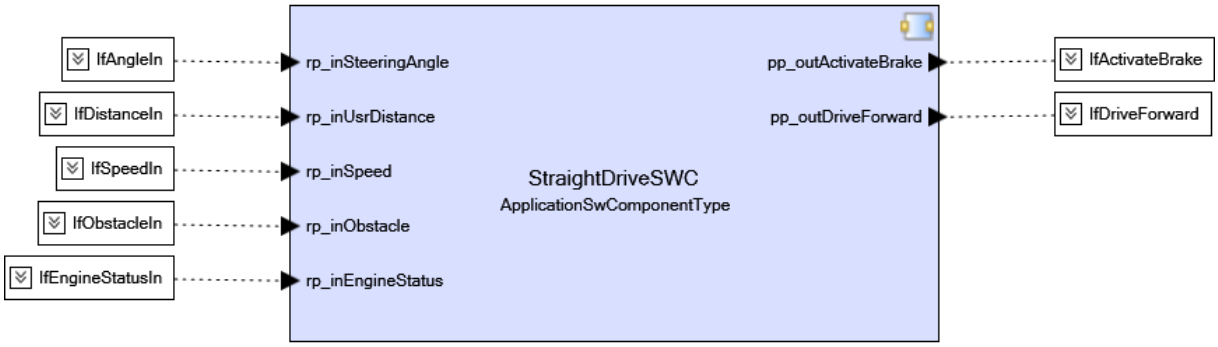

Figure 5.9: StraightDrive SWC Prototype.

critical functionalities within the SWC. Above is the developed prototype SWC, where all user inputs, SteeringAngle, UsrDistance, Speed, Obstacle, and EngineStatus, are defined as RPort, and action-taking functionalities, DriveForward and ActivateBrake from the SWC, are defined as PPorts. All the assigned interfaces are sender-receivers, and a runnable triggering event is also defined as a timing event 10ms like the main one. The RPorts are ready to read data from tester as per the below RTE API functions:

```
/* Read inputs */
Boolean EngineStatus = Rte_IRead_StraightDriveSWC_rp_inEngineStatus_EngineStatusIn();
Boolean ObstacleDetected = Rte_IRead_StraightDriveSWC_rp_inObstacle_ObstacleIn();
uint16 SpeedOfCar = Rte_IRead_StraightDriveSWC_rp_inSpeed_SpeedIn();
uint32 DistanceOfCarKm = Rte_IRead_StraightDriveSWC_rp_inUsrDistance_DistanceIn();
uint8 AngleOfCar = Rte_IRead_StraightDriveSWC_rp_inSteeringAngle_AngleIn();
```

After processing input data by SWC, PPorts will write data as per the below RTE API functions:

```
/* Write Outputs */
Rte_IWrite_StraightDriveSWC_pp_outDriveForward_DriveForward();
Rte_IWrite_StraightDriveSWC_pp_outActivateBrake_ActivateBrake();
```

5.2.2 Simulation Scenarios

Logic implementation for the simulation scenarios is similar to the main SWC. The below functionalities should be tested for the prototype SWC-

- User Input Validation: Validate for all the defined RPorts and ensure that the SWC reads data properly through those ports.

- Engine Status Validation: Check that the engine status setting is working properly. For this simulation engine status is either “on” or “off”. If the engine is off then no other parameter should work.

- Distance Tracking Validation: Check that distance tracking is working based on speed and target distance. For the simulation, the target distance parameter is in kilometers, and the speed is in meter/sec (m/s).

- Obstacle Detection Validation: Check that the car is reading obstacle status properly and taking actions based on detection.

- Forward Driving Validation: Check that the car is driving forward for the provided inputs. Also, for tracking straight drive, the steering angle should be between 0 to 4 degrees.

- Braking Mechanism Validation: Check that the braking mechanism is working properly to achieve the target distance, straight drive, or detect obstacles.

5.2.3 VEOS Simulation

To test on VEOS, need to create a virtual ECU (V-ECU) on SystemDesk, based on the developed architecture along with SWC. After a successful build, there will be a

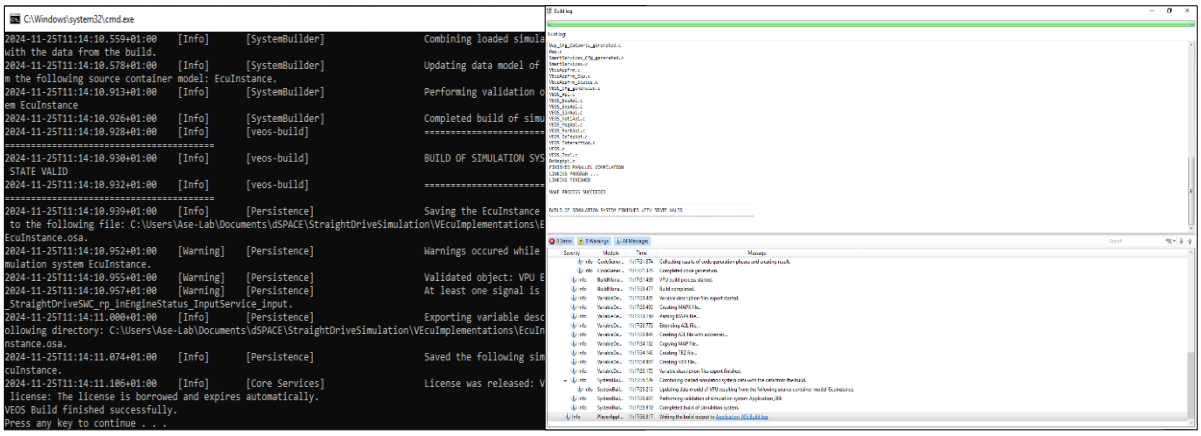

Figure 5.10: Virtual ECU Building.

successful build message. Now, the EcuInstance is created to run on the VEOS player. After that need to import the EcuInstance into the VEOS player. Then, the simulation environment is ready to test functionality for the developed SWC. The test points will be created in the VEOS simulation model based on the developed SWC ports that

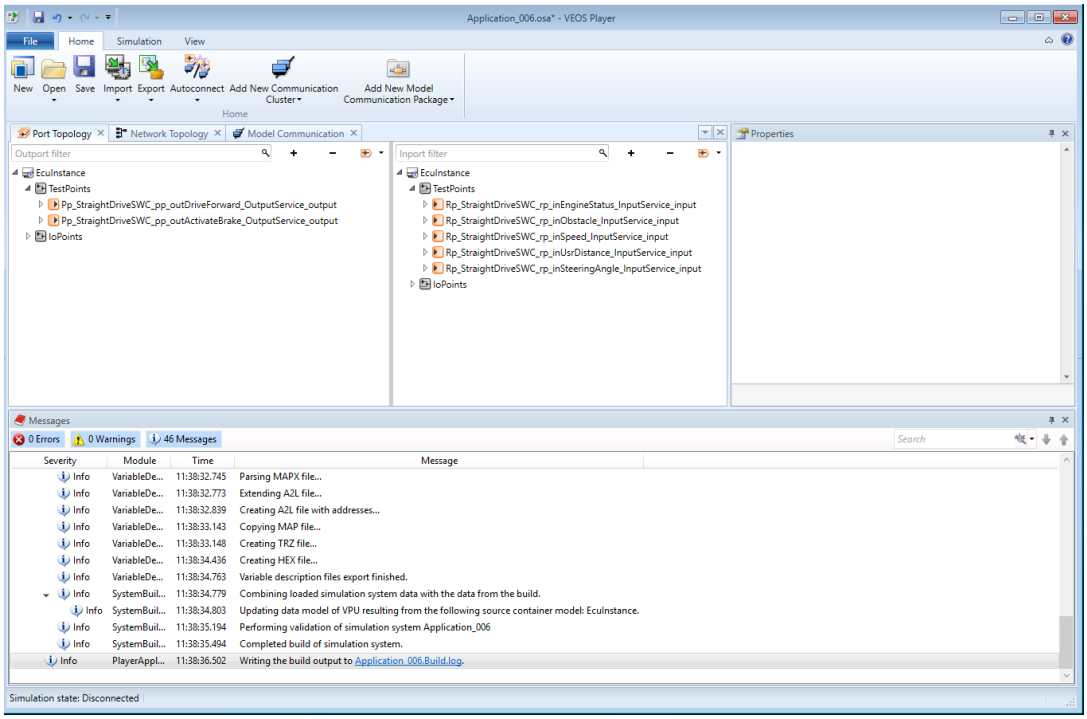

Figure 5.11: VEOS Simulation Test Points.

were created in V-ECU. The simulation was tested on seven different test cases, and the test result is illustrated in the table below.

Table 5.3: SWC Simulation Test Cases.

	Input					Output		Car Message
Test Case	inEngineStatus (on/off)	inObstacle (yes/no)	inSpeed (m/s)	inSteeringAngle (deg)	inUsrDistance (km)	ActivateBrake	DriveForward	
1	Off	n/a	n/a	n/a	n/a	n/a	n/a	Engine Status: OFF, please start the engine.
2	On	n/a	0	0	0	n/a	n/a	Engine ON, but the car is not moving, please accelerate
3	On	n/a	10	0	0	yes	no	Distance threshold reached. Activating Brake
4	On	n/a	10	0	1	yes	yes	1. Activating drive forward 2. Distance threshold reached. Activating Brake after 1000m/1km and 100 sec
5	On	n/a	10	5	1	yes	no	The angle of the car is not within 0 to 4 degrees. Activating brake
6	On	n/a	51	4	1	yes	no	The speed of the Car exceeds 50 m/s. Activating brake
7	On	yes	10	0	1	yes	no	Obstacle detected!Activating Brake

The above table shows that all the test cases were passed successfully. Thus, the SWC prototype was successfully tested in the simulation environment before being implemented in the real-time environment. The developed SWC integration will take place in the next sub-chapter.

5.3 System Integration

Since the SIL test has been completed and passed successfully, the SWC, which was already validated in sub-chapter 5.1.5, can now be integrated into EB Tresos Studio. To do so, need to export AUTOSAR from dSPACE SystemDesk, which will generate a new *.arxml file. The old *.arxml file needs to be replaced by the newly generated file with StraightDriveSWC in

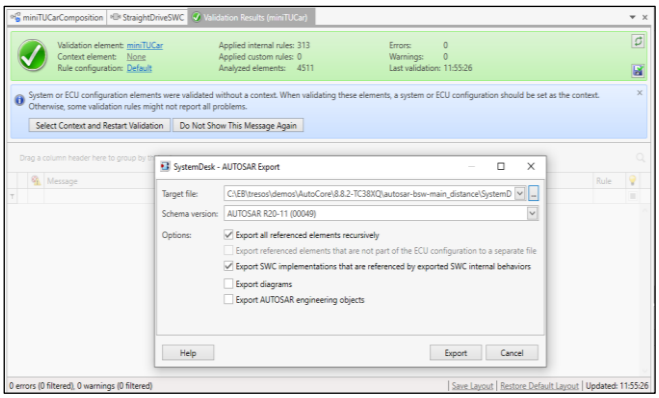

Figure 5.12: AUTOSAR Export from dSPACE.

the same directory. Since the *.arxml file is generated, it's time to go for EB Tresos studio integration, which will be in the next sub-section.

5.3.1 Integration in Tresos Studio

EB Tresos Studio configuration file holds the full AUTOSAR architecture modules along with newly modified SWCs and RTE. Since the new application SWC "StraightDriveSWC" was added to the SystemDescription file, therefore need to be integrated for the regeneration project. For this purpose, need to run an importer from the im – and exporters manager on "miniTUCar_SystemDesc_Imp." After successful importation, there will be a successful message with no error.

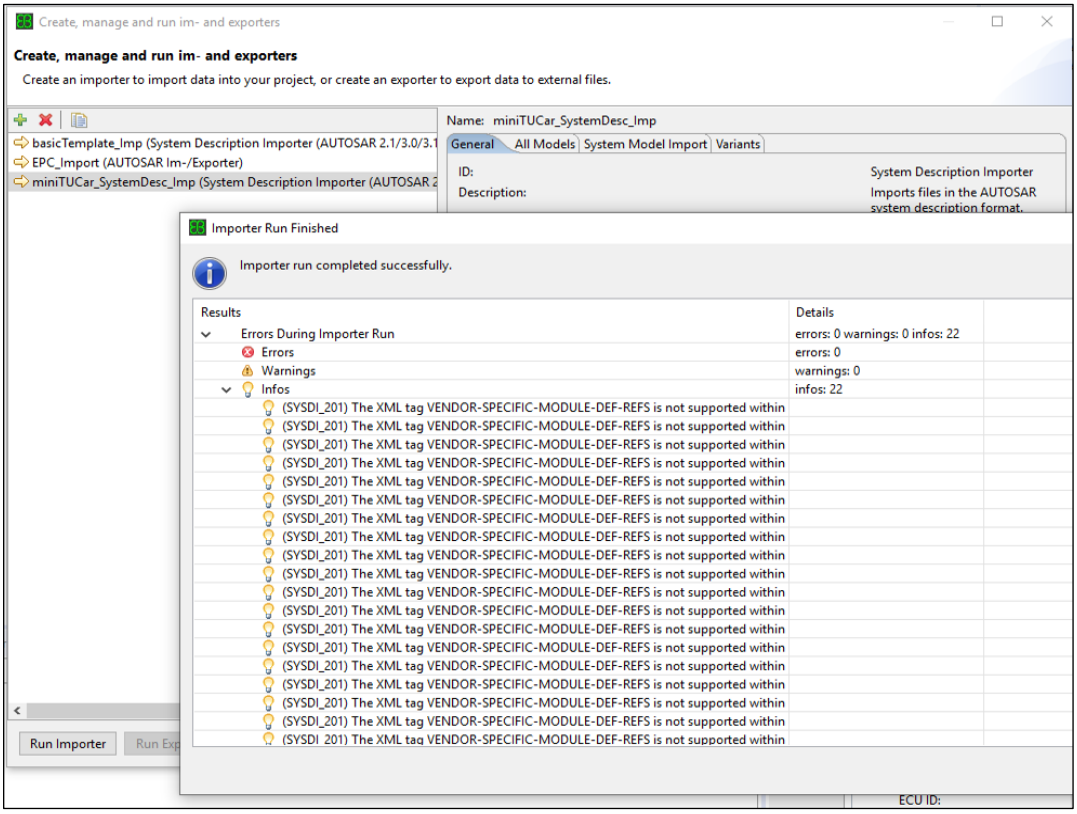

Figure 5.13: Run Importer in EB Tresos Studio.

After successful importation, need to map the unmapped RTE events from the Rte Editor. Since a new timing event was created for the application SWC, "StraightDriveSWC," therefore there will be an unmapped event. This timing event needs to be mapped into the appropriate timing event list. After that, from the project->unattended wizard directory, need to execute MultiTask_RunFullimport to import supporting elements within Tresos Studio. This will complete the importation process.

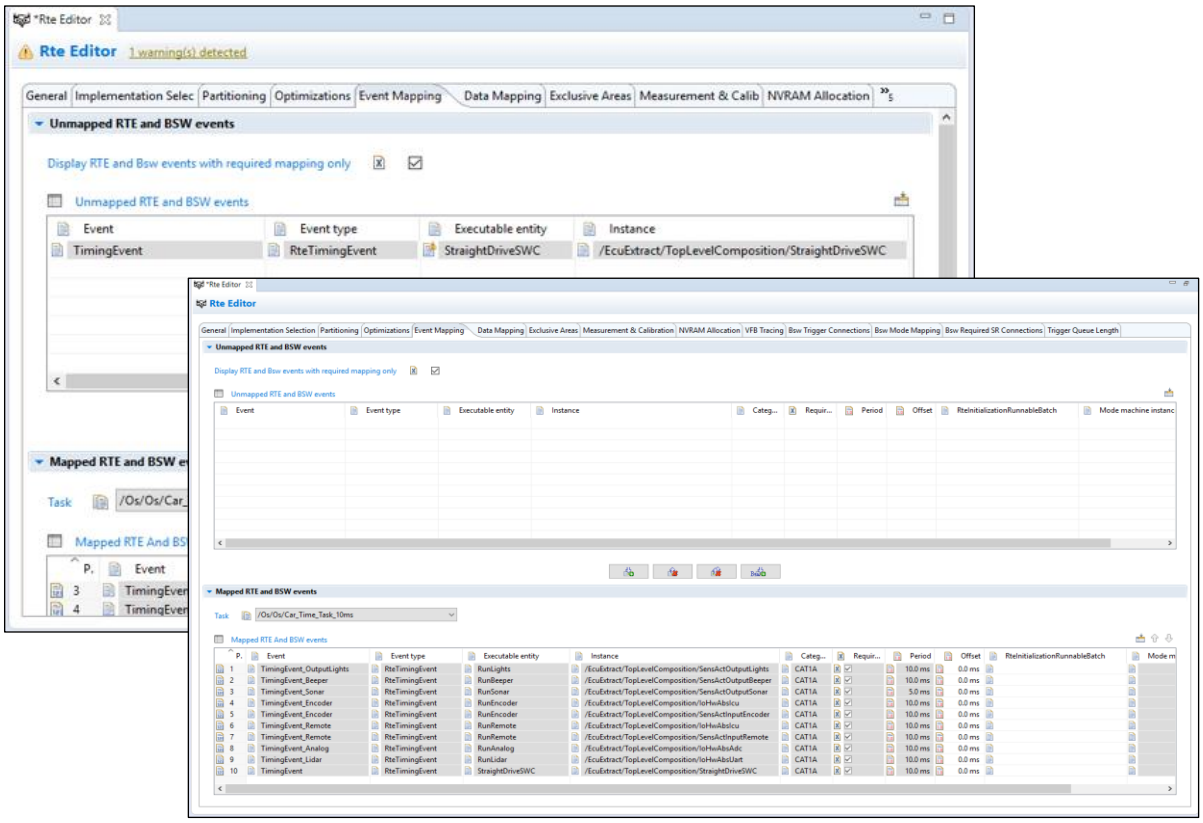

Figure 5.14: RTE Event Mapping.

After RTE event mapping is done, need to check that the newly created ports are all mapped into the connection editor in Tresos Studio.

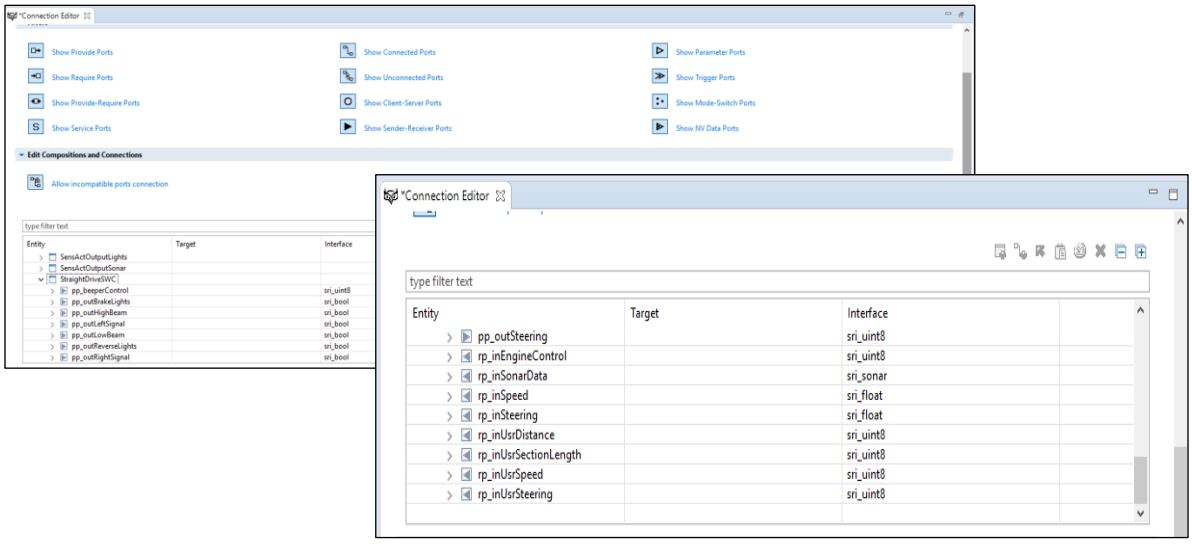

Figure 5.15: Port Mapping in Connection Editor.

5.3.2 System Validation

After completing the integration task in Tresos Studio, need to validate the system. For this, need to verify and generate the project. After successful verification, there should not be any errors in the project error log.

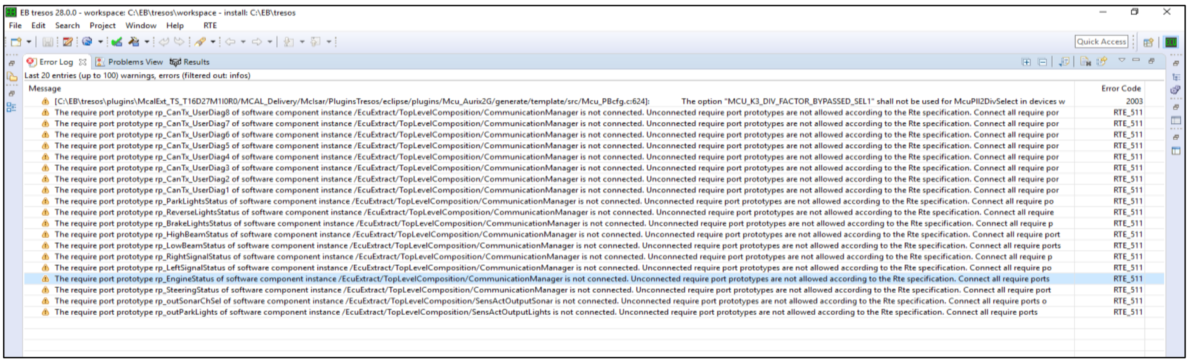

Figure 5.16: Project Generation Error Log.

After verification and generation, the error log shows no error. Only a few warnings are present, those not related to the newly added application SWC. So, the project is ready to compile and generate a *.hex file.

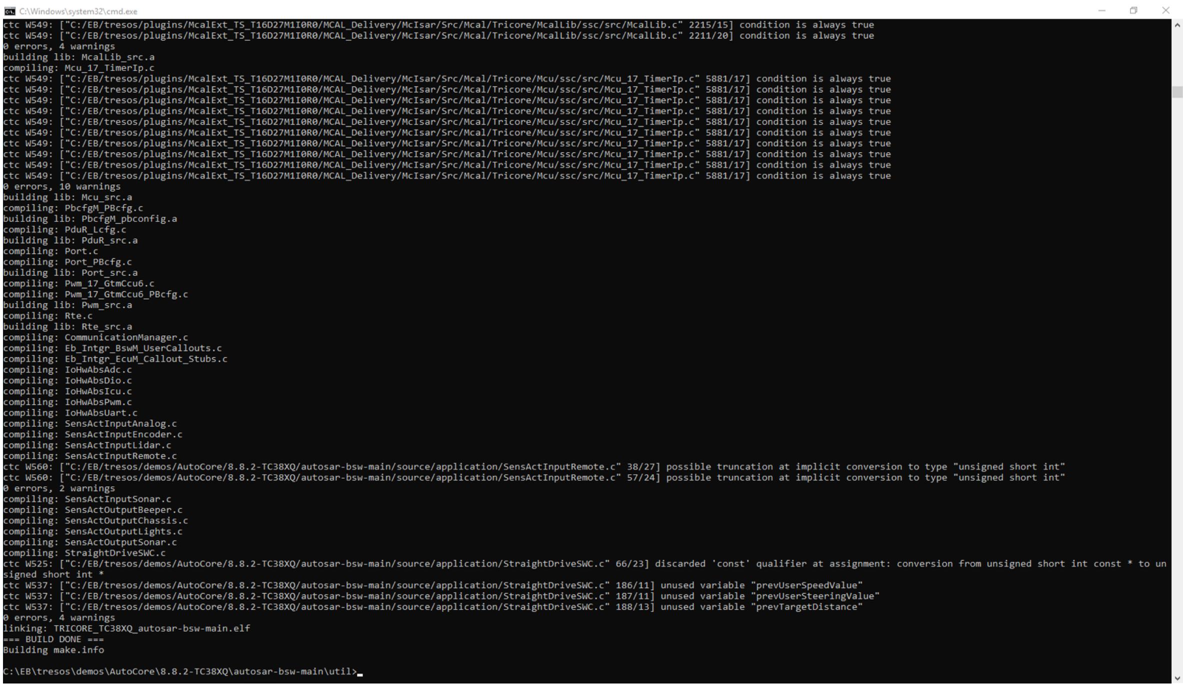

Figure 5.17: Project Compilation.

The project has been successfully compiled by the Tasking compiler, and the build was done without any errors. The “TRICORE_TC38XQ_AtomicStraightDrive.hex” file, which needs to be flashed in the ECU, has been generated into the following directory of the project folder: “AtomicStraightDrive\output\bin\ .”

5.3.3 Setting Up the Test Environment

Since the *.hex file is ready to be flashed, now need to prepare the hardware and software setup to perform an ECU flash. First, need to power on the ECU used for the car, Infenion KIT_A2G_TC387_3V3_TFT. An Infineon DAP miniWiggler is already connected to the ECU through a DAP connector. Then, a connection between the tester (PC) and the ECU was established. From the tester's end, need to connect from Infineon Memtool software to access the ECU. After connecting, the existing configuration was erased from MemTool. Now, the newly generated *.hex file has been imported, which was generated to the project directory: "AtomicStraightDrive\output\bin\ ."

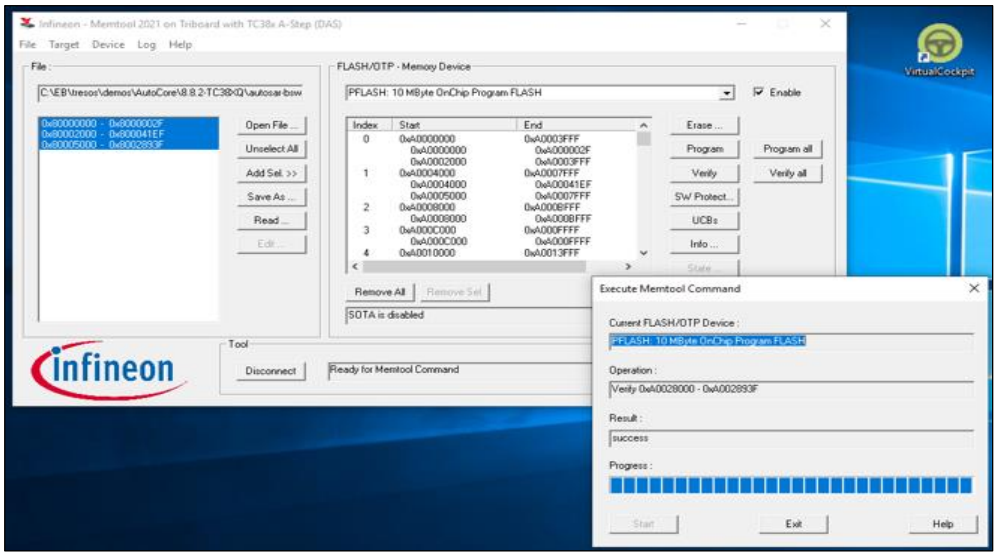

Figure 5.18: Flashing Project into ECU.

After that, need to "Add Select" the components to be flashed into the ECU and program those files. After the program is successful, then need to verify the flashing. There will be a success message on successful verification in the result section. All AUTOSAR components have been integrated into the target ECU of TUCminicar and are now ready to test the functionalities. The CAN bus will be used to send messages from the tester side. For this purpose, the Tiny-CAN II-XL Interface is connected to the ECU by using a 9-pin connector on the ECU side and a USB connector to the tester side (PC). For testing from the tester, the CAN message format below will be used for the project testing:

Table 5.4: CAN Message Format.

Can_Id	DLC	Data							
		rp_inEngineControl	rp_inUsrSpeed	rp_inUsrSteering	rp_inUsrDistance	rp_inUsrSectionLength	Unused	Unused	Unused
0x05A	8	d0	d1	d2	d3	d4	d5	d6	d7

CAN Message Details:

rp_inEngineControl (d0): Send engine control for car, 0 – engine off, 1 – engine one, other value -invalid.

rp_inUsrSpeed (d1): Duty value for speed, 101 – 200: Forward, 100: Neutral, 00-99: Reverse.

rp_inUsrSteering (d2): Duty value for steering angle, 200 - 101: Left direction, 100: Neutral, 00 - 99: Right direction.

rp_inUsrDistance (d3): Number of sections the car wants to travel, 1,2, 3,….and so on.

rp_inUsrSectionLength (d4): Section length, each one atomic section, 1=200mm, 2=400mm, 3=600mm, …and so on.

(d5-d7): Unused.

Two different batteries power the car: one for the control board and the other for the motors. Now, the full hardware and software side is ready to test the car's functionality.

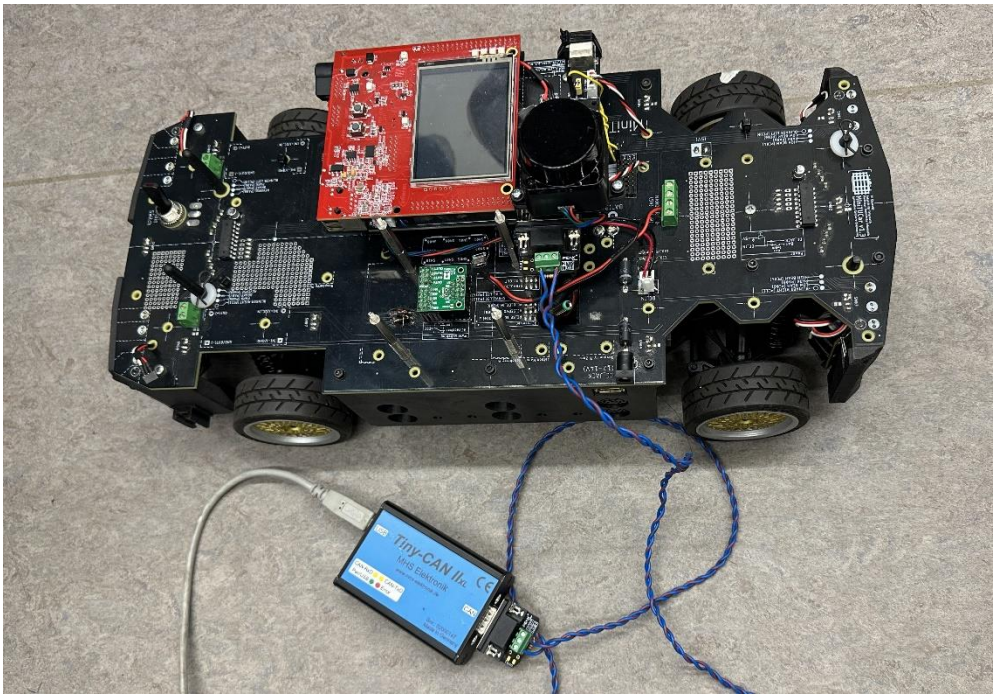

Figure 5.19: TUCminiCar Connected to the Tester.

5.4 System Test

System testing ensures that the components work smoothly and interact with each other according to the developed logic. This is the main section for verifying the whole developed system. The system test processes have been divided into three major sections, which will be discussed in the following subsections: Unit Testing (5.4.1), Integration Testing (5.4.2), and End-to-End Testing (5.4.3). In this chapter, test processes will be described, and then there will be a chapter on the outcome results, where the outcome results will be illustrated.

5.4.1 Unit Testing

The unit section focuses on validating each method implemented, that is, each component within the SWC testing. The unit testing is basically for the different RPort defined in the SWC, where need to check that the SWC is taking input for each defined port. The functionality of the prototype SWC was already tested on the SIL in Chapter (5.2) and now needs to be checked on the hardware. For unit testing, the criteria need to be tested as per the below-

Engine Test: For engine testing, whether the input for controlling the engine is working or not needs to be tested on three different test cases, and the criteria are mentioned in the table below.

Table 5.5: Engine Unit Test Criteria.

Test Case	CAN Msg (uint8)	Expected Operation
1	0	Engine Off, Car is in neutral state, no other input should work
2	1	Engine On: Car is in neutral state, ready to take new inputs
3	any other (e.g., 2)	Invalid engine control input, no operation

To test the engine status functionality, the below CAN messages are sent to the ECU from the tester, and the output status result is presented in the result section (6.1.1).

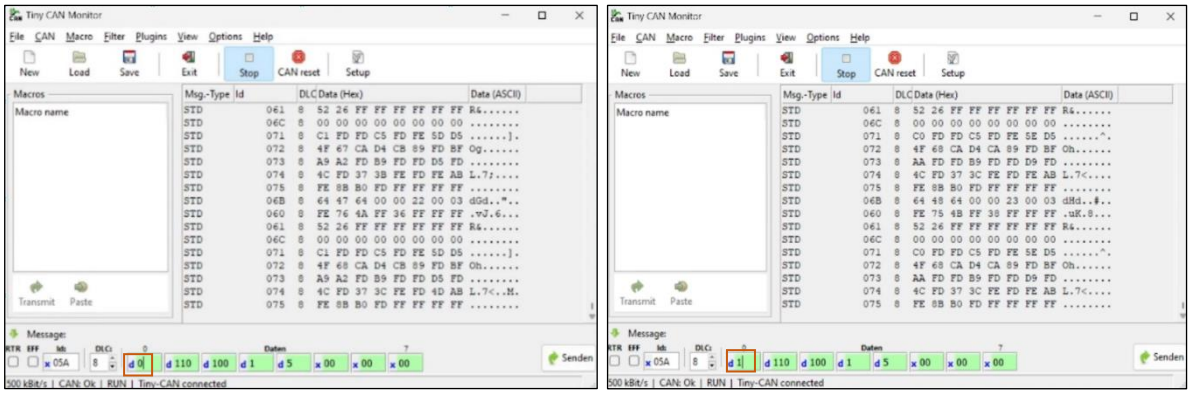

Figure 5.20: CAN Message for the Engine Status Test.

Speed Test: For speed testing, the car needs to perform the three basic operations mentioned in the table below. The duty cycle reference value for the speed is defined in the “miniTUCar_BSW_v0.29 Project Documentation” document. [52]

Table 5.6: Speed Unit Test Criteria.

Test Case	CAN Msg (Duty Value)	Expected Operation
1	200	Car should operate at full speed in forward direction
2	100	Car should be in the Neutral state
3	000	Car should operate at full speed in the reverse direction

To check speed tests within the car, Tiny CAN Monitor sends three different CAN messages, as shown below. The test result is disclosed in the result section (6.1.1).

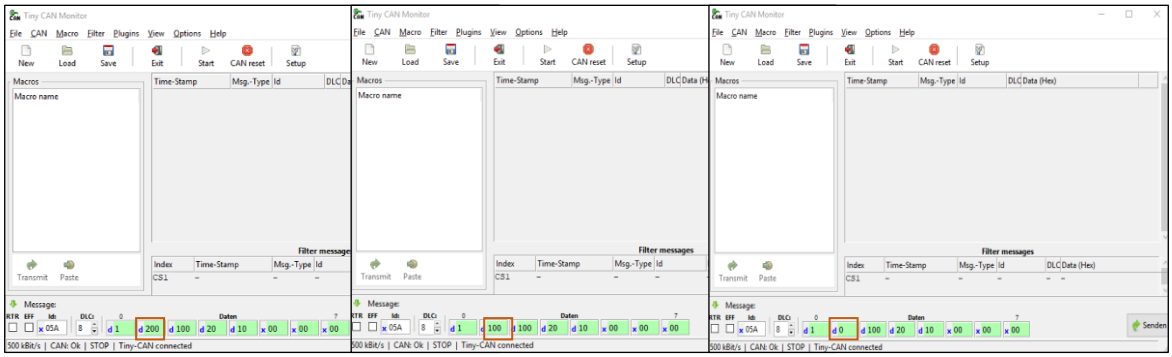

Figure 5.21: CAN Message for the Speed Test.

Steering Angle Test: For Steering Angle testing, the car needs to perform the three basic operations mentioned in the table below. The duty cycle reference value for the steering angle is defined in the “miniTUCar_BSW_v0.29 Project Documentation” document.[52]

Table 5.7: Steering Angle Unit Test Criteria.

Test Case	CAN Msg (Duty Value)	Expected Operation
1	200	The steering Angle should rotate Max-left position
2	100	Steering should be in a Neutral position
3	0	The steering Angle should rotate Max-right position

For the steering angle tests within the car, Tiny CAN Monitor sends three different CAN messages, as shown below. The test result is presented in the result section (6.1.1).

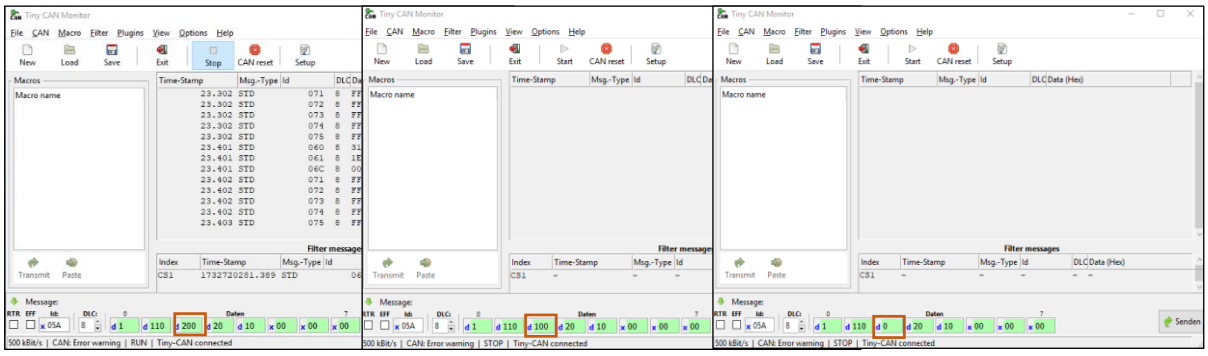

Figure 5.22: CAN Message for the Steering Angle.

Obstacle Detection Test: For obstacle detection unit test, need to check whether the sensor data is being read by the SWC successfully. It is also need to test the range of the sensor area, which is currently set to 200mm in both the forward and reverse directions.

Target Distance Input Test: For the distance input test, it is required to take both input-related distances, rp_inUsrDistance defines how many atomic sections want to

drive, and rp_inUsrSectionLength provides data about section length. Each atomic section length is 200mm, which is calculated in chapter (2.3). For driving one section of 1000mm (1 meter), need to input rp_inUsrSectionLength as 5, which provides 200mm×5 = 1000mm. Or it can be done as five sections from rp_inUsrDistance, want to drive one atomic section (200mm) in each section.

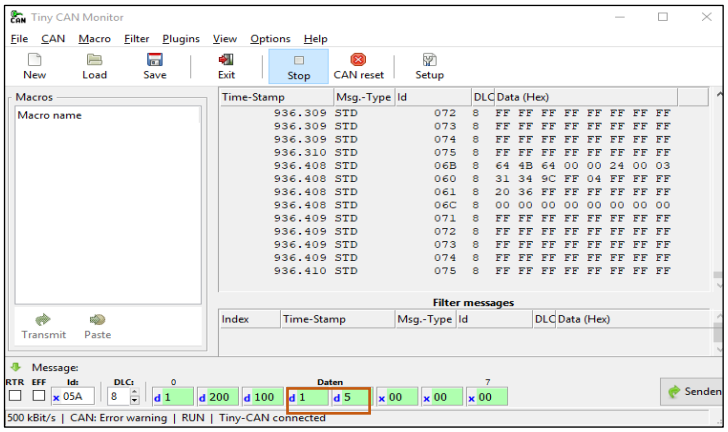

Figure 5.23: Target Distance Input CAN Message.

5.4.2 Integration Test

After completing the unit test, it is time to begin the integration test. This test is an enhanced method to verify the system since it focuses on the integration between different software components (SWCs) and RTE. For the thesis topic, need to check how the other SWCs are interacting with the application SWC developed within the architecture. For the integration test, the following criteria need to be tested:

Travel Distance Test: In the unit test section, the input for distance input data was measured and found to be accurate as per input data validation. Now, it is necessary to test how the SWC named “SensActOutputChassis,” which controls the DC motor unit, reacts. For this test, have to check the cumulative distance covered by the TUCminiCar in response to the target distance. The travel distance is checked for

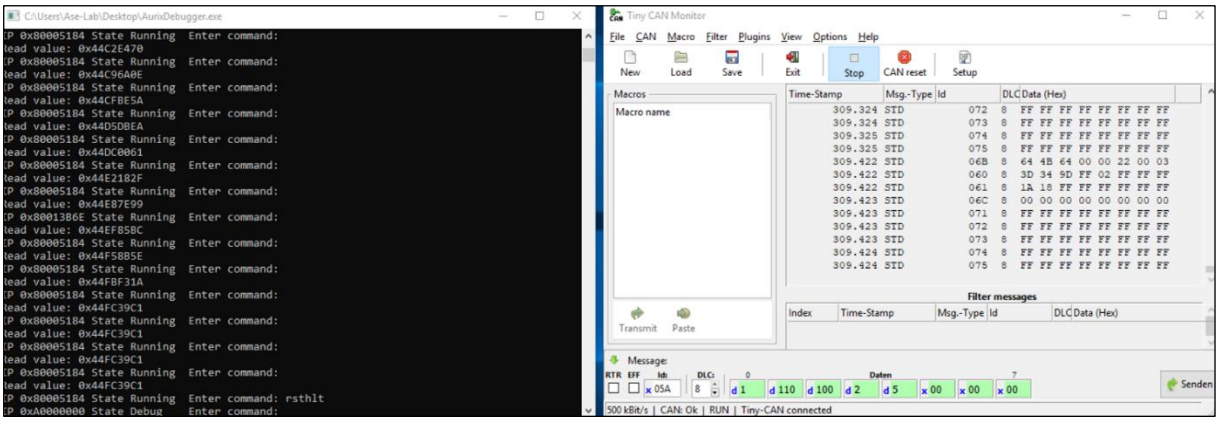

Figure 5.24: CAN Message and Memory Read for Travel Distance Test.

having five atomic sections (5 × 200mm = 1000mm) and two sections of road (2 × 1000mm), so in total, 2 meters of path length. For this purpose, a corresponding CAN message was sent, and distance was read from the memory address assigned as the global variable “float32 cumulativeDistance”. Test results will be analyzed later in the result section (6.1.2).

Straight Driving Test: To verify the car's straight-driving capabilities, the steering angle needs to be neutral for the traveled distances. Like the DC motor, the servo motor for steering control is also part of the “SensActOutputChassis” SWC, which needs to test how it reacts with the interacting “StraightDriveSWC” application SWC. To test this, a CAN message was sent to operate the car for 40 meters, with a speed duty cycle of 110, and keeping the steering angle in a neutral position as per the CAN message pictured below. After traveling the whole distance, steering angle data was

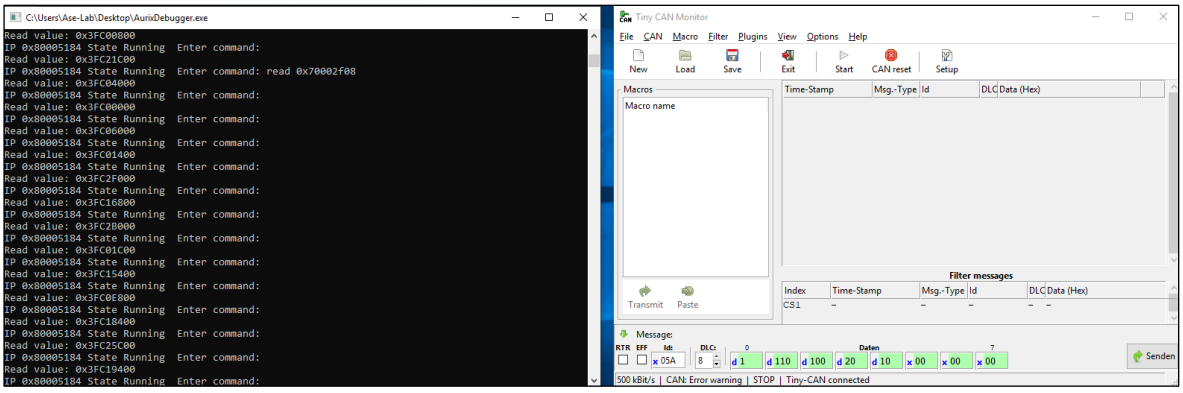

Figure 5.25: CAN Message and Memory Read for Straight Drive Test.

read from the memory address, and the result will be analyzed in the result section (6.1.2).

Braking Mechanism Test: The braking mechanism should work for the two main arguments: Travel Distance Reached and Obstacle Detected. The both cases are described below-

- **Braking due to Target Distance Reached**

The automatic braking should work when the target distance is reached in both forward and reverse directions. The Can messages were sent to test both directions as per below, and the result has been analyzed in the result section (6.1.2).

Forward Direction: In the forward direction, the braking mechanism was tested for five sections with five atomic lengths, which is 5000 mm (5 meters) in total distance. The speed duty value was 105, which is equivalent to 180mm/s. After the car reached the travel distance, then the cumulative distance data was read from the memory address.

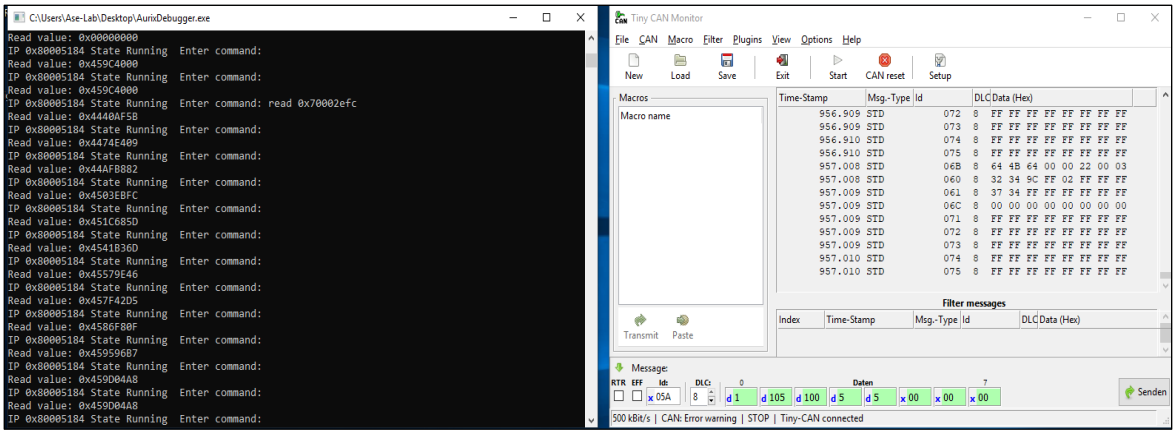

Figure 5.26: CAN Message and Memory Read for Forward Drive Braking.

Reverse Direction: In the reverse direction, the braking mechanism was also tested for five sections with five atomic lengths, which is 5000 mm (5 meters) in total distance. The speed duty value was 99, which is equivalent to 140mm/s. After braking was activated, traveled distance data was checked from ECU memory.

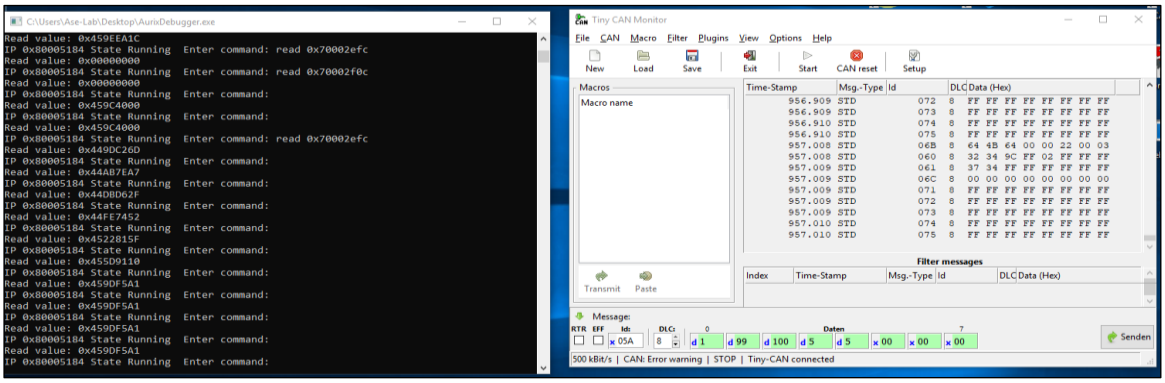

Figure 5.27: CAN Message and Memory Read for Reverse Drive Braking.

- **Braking due to Obstacle Detected**

For obstacle detection, the braking mechanism also needs to be validated for both forward and reverse directions since both side sonar sensors are activated.

Forward Direction: In the forward direction, the braking mechanism was tested for a speed duty value of 150, which is equivalent to 630mm/s. The CAN message was sent

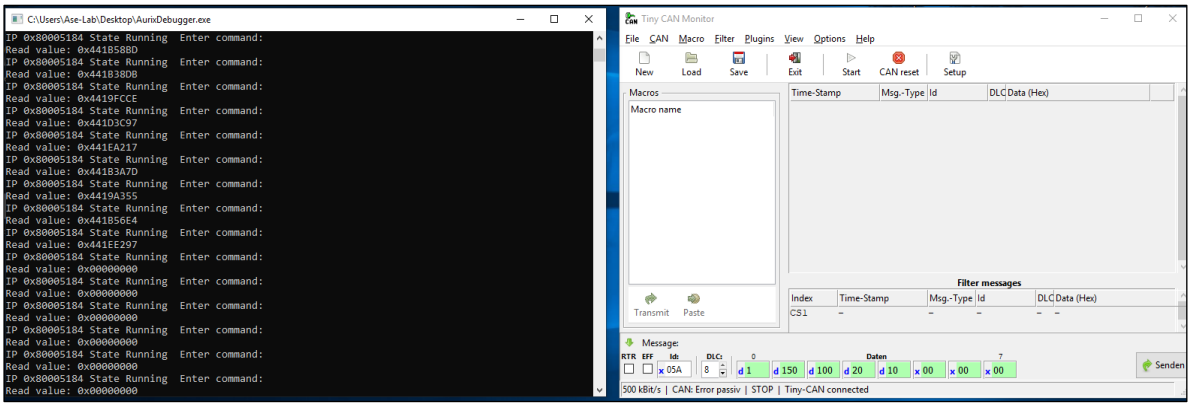

Figure 5.28: CAN Message for Forward Braking Obstacle Detection.

accordingly, and the memory read value was checked to confirm speed during obstacle detection. The result analysis is in the result chapter (6.1.2).

Reverse Direction: In the reverse direction, the obstacle detection and braking mechanism was tested for 64 duty cycles of speed, which is equivalent to 490 mm/s, and the following CAN message was sent accordingly. After the obstacle is detected

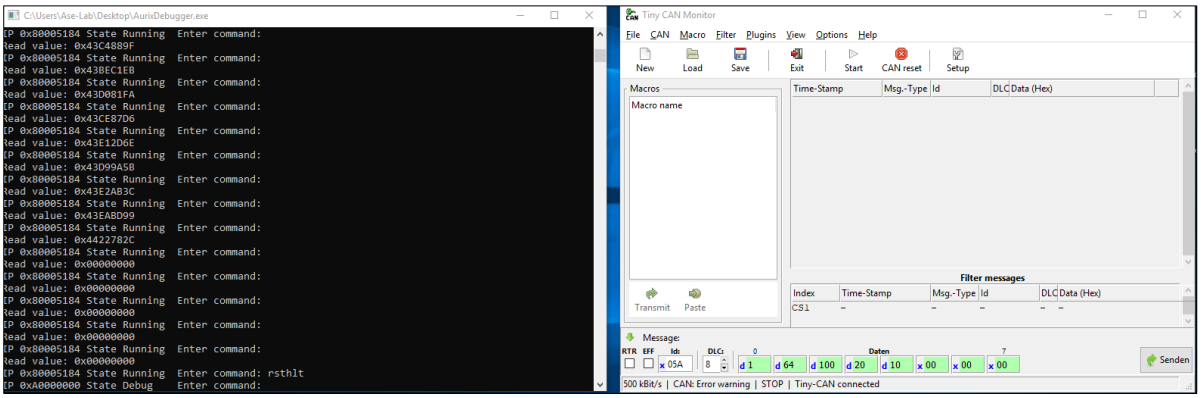

Figure 5.29: CAN Message for Forward Reverse Obstacle Detection.

in the car's reverse direction, speed data is checked in memory to verify. The result has been discussed in the result section (6.1.2).

Auditory Actuator Test: The Beeper is installed in the TUCminiCar, an auditory actuator that is part of the “SensActOutputBeeper” SWC. The Beeper is integrated into the application SWC “StraightDriveSWC” for obstacle detection. Now, need to verify whether the Beeper is activated after detecting an object. For this purpose, CAN messages are sent to drive forward and backward and tested by placing an object in front of the car. The result has been disclosed in the result chapter (6.1.2).

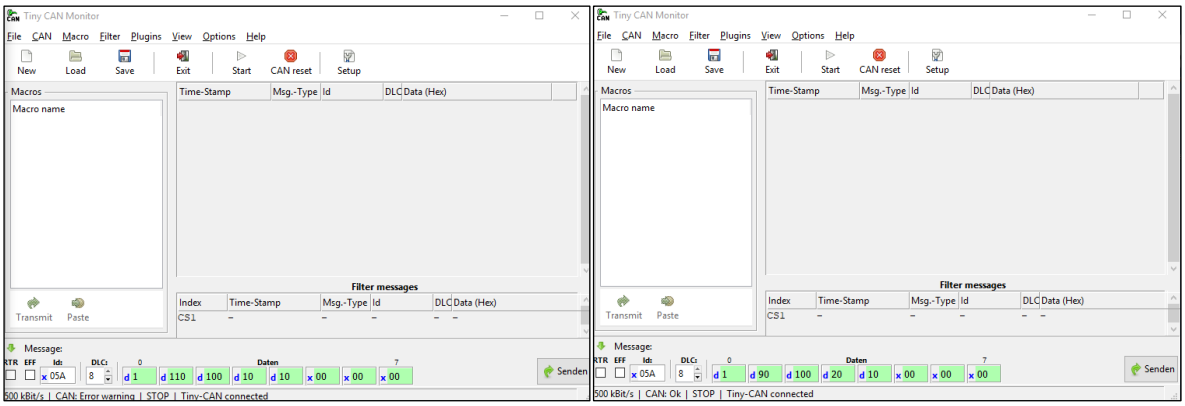

Figure 5.30: CAN Message to Test Beeper.

Visual Actuator Test: Different visual actuators are integrated within the TUCminiCar. All the lights are controlled by the “SensorActuatorSwComponentType” SWC “SensActOutputLights.” All the required lights are integrated into the application SWC “StraightDriveSWC.” To check the functionalities of these lights, different CAN

messages are sent as below, and the test result has been analyzed in the result section (6.1.2)

Table 5.8: CAN Message for Visual Actuator Test.

Criteria	Test Case	Effecting CAN Msg Input	CAN Msg (uint8)
LeftSignal	1	rp_inUsrSteering (d2)	200
	2	rp_inUsrSteering (d2)	100
	3	rp_inUsrSteering (d2)	0
RightSignal	1	rp_inUsrSteering (d2)	200
	2	rp_inUsrSteering (d2)	100
	3	rp_inUsrSteering (d2)	0
ReverseLights	1	rp_inUsrSpeed (d1)	100
	2	rp_inUsrSpeed (d1)	99
HighBeam	1	rp_inUsrSpeed (d1)	150
	2	rp_inUsrSpeed (d1)	140
LowBeam	1	rp_inUsrSpeed (d1)	100
	2	rp_inUsrSpeed (d1)	110
BrakeLights	1	rp_inUsrSpeed (d1)	110
	2	rp_inUsrSpeed (d1)	Brake Activated

The above CAN message table is based on the CAN message format already explained (Table 5.4) in the chapter (5.3.3). The engine status must be on (1), and other fields are subject to change for each visual actuator testing. The result analysis has been placed in the result section (6.1.2).

5.4.3 End-to-End (E2E) Testing

The End-to-End (E2E) testing is the overall testing or the acceptance test for the project. The main project goal needs to be validated in this test, which is validation for different atomic straight-driving scenarios. Depending on different speeds, distances, and obstacle detection, for each criterion, this test was conducted for five test cases, and those are described below-

Scenario 1: Forward Drive - Fixed Speed with Varying Distance Coverage

Five different CAN messages were sent for fixed-speed duty cycle 105, with target distances of 1m, 2m, 3m, 4m, and 5m. The CAN message format is as per the defined standard mentioned in chapter (5.3.3), where d0 position for engineControl, d1 is for Speed, d2 is for steeringControl, d3 is for numberOfSection, d4 is for sectionLength, and d5-7 are unused. Engine status is always “On” here, and the steering duty cycle is kept at “100” to maintain a straight line. The target distance was fixed as five atomic lengths (5 × 200mm = 1000 mm) by varying the number of sections (1/2/3/4/5), and CAN messages were sent as per the table below. (Note: All the sent messages are in decimal format)

Table 5.9: CAN Message for Fixed Speed, Target Distance Change.

Test Case	Can Id	DLC	d0	d1	d2	d3	d4	d5	d6	d7
1	x05A	8	1	105	100	1	5	0	0	0
2	x05A	8	1	105	100	2	5	0	0	0
3	x05A	8	1	105	100	3	5	0	0	0
4	x05A	8	1	105	100	4	5	0	0	0
5	x05A	8	1	105	100	5	5	0	0	0

The test result observation has been made after checking the memory address value for the cumulative distance covered for each test case, which is presented in the result section (6.1.3).

Scenario 2: Forward Drive - Fixed Speed, Different Atomic Sections

In the previous test scenario (1), it was observed that as the target distance decreases, accuracy also decreases. To analyze this further, path length decreases more in this test section, keeping the speed duty cycle the same (105) as before. CAN messages were sent for having the same speed but different atomic sections. Engine status remains “On” here always, and the steering duty cycle is kept at “100” to maintain a straight line. The target distance was varied as one atomic path length each (1 × 200mm = 200 mm, 2 × 200mm = 400 mm, 3 × 200mm = 600 mm, 4 × 200mm = 800 mm, 5 × 200mm = 1000 mm,) by keeping the of sections fixed (1), and CAN messages were sent as per the table below.

Table 5.10: CAN Message for Fixed Speed, Atomic Section Length.

Test Case	Can Id	DLC	d0	d1	d2	d3	d4	d5	d6	d7
1	x05A	8	1	105	100	1	1	0	0	0
2	x05A	8	1	105	100	1	2	0	0	0
3	x05A	8	1	105	100	1	3	0	0	0
4	x05A	8	1	105	100	1	4	0	0	0
5	x05A	8	1	105	100	1	5	0	0	0

The test result is observed in the result section (6.1.3)

Scenario 3: Reverse Drive - Fixed Target Distance, Different Speed

Likewise, for the forward speed, need to analyze the reverse direction as well. For this purpose, CAN messages were sent for a fixed atomic target distance, but this time, they were checked for different speed values. The target distance was fixed as five atomic path lengths (5 × 200mm = 1000 mm) and the number of sections was fixed (1), and CAN messages were sent as per the table below.

Table 5.11: CAN Message for Fixed Atomic Distance, Different Speed in Reverse.

Test Case	Can Id	DLC	d0	d1	d2	d3	d4	d5	d6	d7
1	x05A	8	1	90	100	1	5	0	0	0
2	x05A	8	1	80	100	1	5	0	0	0
3	x05A	8	1	70	100	1	5	0	0	0
4	x05A	8	1	60	100	1	5	0	0	0
5	x05A	8	1	50	100	1	5	0	0	0

The test result is presented in the result section (6.1.3)

Scenario 4: Expected speed Vs. Actual Speed

From the last test, it is observed that as speed increases, accuracy is lower. To analyze this further, three different speed data were selected (duty cycle—110, 150, 200), and a memory read was performed ten different times to check fluctuation in actual speed. The sample CAN message was sent to test the car's speed for three different duty cycles, as shown below. Since the observation was made with respect to speed, distance was not counted this time.

Table 5.12: CAN Message for Different Speed.

Test Case	Can Id	DLC	d0	d1	d2	d3	d4	d5	d6	d7	Remarks
1	x05A	8	1	110	100	10	10	0	0	0	10 Times memory read
2	x05A	8	1	150	100	20	10	0	0	0	10 Times memory read
3	x05A	8	1	200	100	20	10	0	0	0	10 Times memory read

The memory read was done ten times for each CAN message, and the test result is presented in the result section (6.1.3)

6 Results and Evaluation

This chapter presents the overall research project outcome by presenting the results of different test cases and evaluating them. The chapter is divided into two main sub-chapters: one is for the Result (6.1), and the other is for the Evaluation (6.2) of the outcome results.

6.1 Result

The result section presents all the result outcomes from the different test scenarios. Based on the System Test (5.4) subchapter, this chapter reflects all the results from those test cases. Like the test section, the result section is also divided into three main sub-sections: Unit Test Result (6.1.1), Integration Test Result (6.1.2), and End-to-End (E2E) Test result (6.1.3).

6.1.1 Unit Test Result

In this result section, the results from the sub-section (5.4.1) are described. Different functionalities were tested for unit tests, like Engine, Speed, Steering Angle, Obstacle Detection, and Target Distance Input. The result for all the combined sections is illustrated in the below table-

Table 6.1: Unit Test Result.

Criteria	Test Case	CAN Msg (uint8)	Status Validation	Result
Engine Test	1	0	Engine Off	Passed
	2	1	Engine On	Passed
	3	2	No Effect on Car	Passed
Speed Test	1	200	Car traveled forward direction	Passed
	2	100	Car Stopped/No Movement	Passed
	3	0	Car traveled reverse direction	Passed
Steering Angle Testing	1	200	Steering Angle Rotate full left	Passed
	2	100	Car directed straight	Passed
	3	0	Steering Angle Rotate full right	Passed
Obstacle Detection Testing	1	n/a	Obstacle detected within 200mm distance	Passed
Target Distance Input Test	1	d3:001, d4:005	Target distance read from memory, 1000mm verified	Passed

Unit Test Result Summary: All other values, except target distance input, are checked by visual inspection from the car, and for the target distance, input data verified from the memory address (0x70002f0c) and found as below:

“IP 0x80005184 State Running Enter command: read 0x70002f0c “(memory address)” Read value: 0x447A0000”

Here, in the memory location, the HEX value is written, Hex (0x447A0000) = Float value (1000.0), which is a 1000mm target distance.

6.1.2 Integration Test Result

Integration test results are the representation of different test cases in the sub-chapter (5.4.2). Different functionalities that are dependent on other SWCs were tested, and the results are illustrated below-

Travel Distance Test Result: For the travel distance test, a CAN message was sent for traveling 2 meters (2000 mm) distance. After analyzing, below data found from memory location:

Variable for checking target distance: float32 targetDistance,
Variable for checking traveled distance: float32 cumulativeDistance,
Memory Address for targetDistance: 0x70002f0c, read value Hex (0x44FA0000),
Memory Address for cumulativeDistance: 0x70002efc, read value Hex (0x44FC39C1).

Table 6.2: Travel Distance Test Result.

Speed Duty Value	Expected Speed in mm/s	Target Distance mm (Converted Float Value)	Traveled Distance in mm (Converted Float Value from Hex)	Test Result	Travel Deviation (mm)	Accuracy Obtained (%)
110	230 mm/s	2000 mm	2017.8 mm	Passed	17.8 mm	99.11

The result above shows that the car traveled 17.8 mm more than the target distance. The accuracy of the travel distance test result is pretty much higher, but still need to check further data to analyze it more during the E2E test.

Straight Driving Test: For the straight driving test, a CAN message was sent for traveling 40 meters (40000 mm) distance straight path. After analyzing, the below data was found for the steering angle from the memory location:

Variable for checking steering angle: float32 globalCarSteeringValue,
Memory Address for globalCarSteeringValue: 0x70002f08, read value Hex (0x3FC19400).

Table 6.3: Straight Drive Test Result.

CAN Msg (Steering Duty Value)	Expected Steering Angle	SteeringPot value for Neutral (in mV)	Actual SteeringPot in mV (Converted Float Value from Hex)	Test Result	Voltage Deviation (mV)	Accuracy Obtained (%)
100	0 deg	1500 mV	1.5123 V = 1512.3 mV	Passed	12.3 mV	99.18

The result above shows that, after traveling 40 meters of distance, the steering potentiometer voltage was found to be 12.3 mV more than the required voltage. The accuracy (99.18 %) for the steering angle of the straight drive test result is pretty much higher, but it still needs to be analyzed further during the E2E test. (The standard for the steering angle to maintain a neutral position is to have 1.5 volts or 1500 mV of SteeringPot value, which is found in the miniTUCar_BSW_v0.29 Project Documentation). [52]

Braking Mechanism Test Result: The braking mechanism was tested for two main criteria, and the results are illustrated below-

- **Braking due to Target Distance Reached Result**

CAN messages were sent in both the forward and reverse directions to travel 5000 mm at 105 and 099 duty cycles value of speed, respectively. The car was stopped after braking was activated when the target distance was covered. The more detailed data has been represented below-

Variable for checking target distance: float32 targetDistance,
Variable for checking traveled distance: float32 cumulativeDistance,
Memory Address for targetDistance: 0x70002f0c, read value Hex (0x459C4000),
Memory Address for cumulativeDistance: 0x70002efc, read value Hex (Forward: 0x459D04A8, Reverse: 0x459DF5A1).

Table 6.4: Braking on Target Distance Covered Result.

CAN Msg (Speed Duty Value)	Expected Speed in mm/s	Target Distance mm (Converted Float Value from Hex)	Traveled Distance in mm (Converted Float Value from Hex)	Test Result	Travel Deviation (mm)	Accuracy Obtained (%)
105 (Forward)	180 mm/s	5000 mm	5024.58 mm	Passed	24.58 mm	99.5084
099 (Reverse)	140 mm/s	5000 mm	5054.70 mm	Passed	54.70 mm	98.906

The above result table shows that the car stopped automatically after reaching the target distance in both forward and reverse directions. In both cases, accuracy is high, but for comparison, the reverse direction has less accuracy (98.9%) compared to the forward direction (99.5%). In the E2E test, more data will be analyzed for further analysis.

- **Braking due to Obstacle Detected**

For obstacle detection, both forward and reverse directions were checked. CAN messages were sent in both the forward and reverse directions to travel at 150 and 064 duty cycles value of speed, which is equivalent to 630 mm/s and 490 mm/s, respectively. The car was stopped after braking was activated when the obstacle was detected, and speed was neutralized. The more detailed data has been represented below-

Variable for checking travel speed: float32 globalCarSpeedValue,
Memory Address for globalCarSpeedValue: 0x70002f04, read value Hex initial (Forward: 0x441B58BD, Reverse: 0x43F3BA8E), read value Hex after Obstacle detection (Forward: 0x00000000, Reverse: 0x00000000)

Table 6.5: Braking on Obstacle Detection Result.

CAN Msg (Speed Duty Value)	Expected Speed in mm/s	Initial Speed mm/s (Converted Float Value from Hex)	Speed after Obstacle Detection (Converted Float Value from Hex)	Test Result	Accuracy Obtained (%)
150 (Forward)	630 mm/s	621.38 mm/s	0 mm/s	Passed	100
064 (Reverse)	490 mm/s	487.45 mm/s	0 mm/s	Passed	100

The above data, which reads from the memory address for speed after the obstacle detection, shows that the speed is zero both in the forward and reverse direction, which means the car stopped at obstacle detection. Although the obstacle detection test result accuracy is 100%, there is some deviation in actual car speed, which needs to be analyzed with more data in the E2E test.

Auditory Actuator Test Result: To test the beeper, the car was accelerated in both forward and reverse directions. Then, an obstacle was placed in both directions, and the result is illustrated in the table below.

Table 6.6: Auditory Actuator Test Result.

Test Case	CAN Msg (Speed Duty Value)	Expected Speed in mm/s	Expected Beep Sound (Car to Obstacle Distance, mm)	Beeper Status	Test Result	Accuracy Obtained (%)
1	110 (Forward)	230 mm/s	200mm	Yes	Passed	100
2	090 (Reverse)	230 mm/s	200mm	Yes	Passed	100

The above table shows that the obstacle was detected during both forward and reverse driving, and in both cases, the beeper was activated.

Visual Actuator Test Result: For visual actuators testing, in response to different CAN messages, below visual actuators activated pictures are illustrated. Later on, the result table will also include the following:

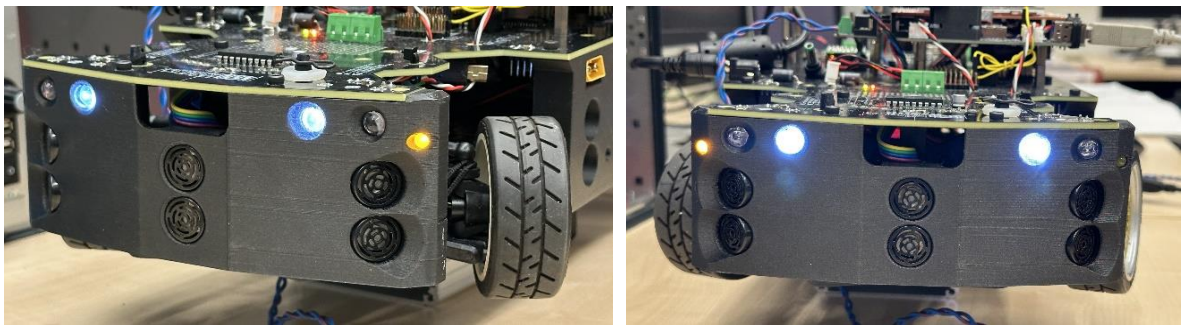

Figure 6.1: Left and Right Signal Light Blinking.

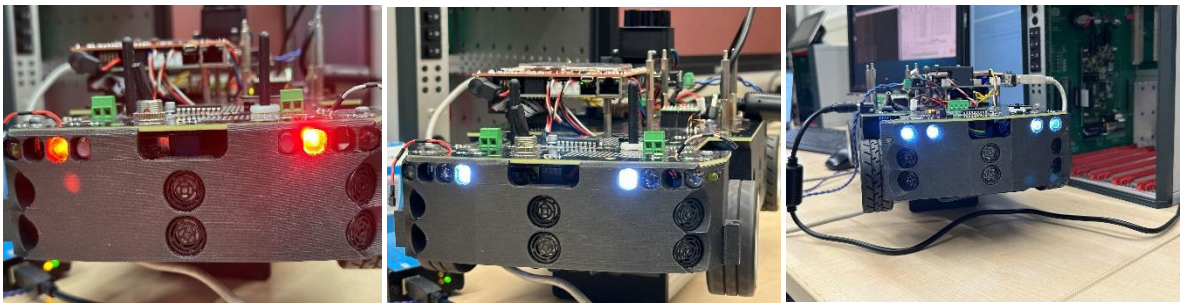

Figure 6.2: Brake Light, Reverse Light, High Beam and Low Beam.

The result table is below based on the test conducted on all the above visual actuators.

Table 6.7: Visual Actuator Test Result.

Criteria	Test Case	Effecting CAN Msg Input	CAN Msg (uint8)	Expected Output	Result	Accuracy Obtained (%)
LeftSignal	1	rp_inUsrSteering (d2)	200	Turned on	Passed	100
	2	rp_inUsrSteering (d2)	100	Off	Passed	100
	3	rp_inUsrSteering (d2)	0	Off	Passed	100
RightSignal	1	rp_inUsrSteering (d2)	200	Off	Passed	100
	2	rp_inUsrSteering (d2)	100	Off	Passed	100
	3	rp_inUsrSteering (d2)	0	Turned on	Passed	100
ReverseLights	1	rp_inUsrSpeed (d1)	100	Off	Passed	100
	2	rp_inUsrSpeed (d1)	99	Turned on	Passed	100
HighBeam	1	rp_inUsrSpeed (d1)	150	Turned on	Passed	100
	2	rp_inUsrSpeed (d1)	140	Off	Passed	100
LowBeam	1	rp_inUsrSpeed (d1)	100	Off	Passed	100
	2	rp_inUsrSpeed (d1)	110	Turned on	Passed	100
BrakeLights	1	rp_inUsrSpeed (d1)	110	Off	Passed	100
	2	rp_inUsrSpeed (d1)	Braking (100)	Turned on	Passed	100

As per the above result table, all the criteria for light testing are fulfilled. There is no CAN message for the brake lights; these lights will be activated once braking is activated, either for an obstacle or for the target distance traveled, and the duty cycle is set to 100 automatically.

6.1.3 End-to-End (E2E) Test Result

End-to-end (E2E) test results represent the overall test, which was done in the sub-chapter (5.4.3). Results for different test cases are presented below-

Scenario 1: Forward Drive - Fixed Speed with Varying Distance Coverage (Result)

For the different CAN messages, different data are extracted from the ECU's memory address and represented below.

Memory Address for targetDistance: 0x70002f0c, read value Hex (Test case: one - 0x447A0000, two - 0x44FA0000, three - 0x453B8000, four - 0x457A0000, five - 0x459C4000),
Memory Address for cumulativeDistance: 0x70002efc, read value Hex (Test case: one - 0x44809DCF, two - 0x44FDFE7A, three - 0x453D4314, four - 0x457B6445, five - 0x459D04A8).

Table 6.8: Test Result for Fixed Speed, Different Target Distance.

Test Case	CAN Msg (Speed Duty)	Expected Speed in mm/s	Target Distance mm (Converted Float Value from Hex)	Traveled Distance in mm (Converted Float Value from Hex)	Test Result	Travel Deviation (mm)	Accuracy Obtained (%)
1	105	180	1000	1028.93	Passed	28.93	97.1
2	105	180	2000	2031.95	Passed	31.95	98.4
3	105	180	3000	3028.19	Passed	28.19	99.06
4	105	180	4000	4022.26	Passed	22.26	99.44
5	105	180	5000	5024.58	Passed	24.58	99.5

After analyzing different target distances for the same speed, it is found that accuracy is still very good, but more car travels have more accuracy than less distance covered.

Scenario 2: Forward Drive - Fixed Speed, Different Atomic Sections (Result)

For testing the different atomic sections, different CAN messages were sent, and different data were extracted from the ECU's memory address, which is represented below.

Memory Address for targetDistance: 0x70002f0c, read value Hex (Test case: one - 0x43480000, two - 0x43C80000, three - 0x44160000, four - 0x44480000, five - 0x447A0000),
Memory Address for cumulativeDistance: 0x70002efc, read value Hex (Test case: one - 0x4366C390, two - 0x43D3B284, three - 0x441E8185, four - 0x444EF54B, five - 0x44809DCF).

Table 6.9: Test Result for Fixed Speed, Different Atomic Section Length.

Test Case	CAN Msg (Speed Duty)	Expected Speed in mm/s	Target Distance mm (Converted Float Value from Hex)	Traveled Distance in mm (Converted Float Value from Hex)	Test Result	Travel Deviation (mm)	Accuracy Obtained (%)
1	105	180	200	230.76	Passed	30.76	84.62
2	105	180	400	423.39	Passed	23.39	94.15
3	105	180	600	634.02	Passed	34.02	94.33
4	105	180	800	827.83	Passed	27.83	96.52
5	105	180	1000	1028.93	Passed	28.93	97.1

After analyzing the above table, it is clear that the less distance the car needs to travel for each message, the less accuracy is. However, it is observed that the amount of deviation added to each test case is almost the same since the smaller amount has a bigger effect on the percentage calculation, thus having less accuracy for the smallest atomic section length (200 mm), which is a really small amount of traveling distance but still having 84.62% of accuracy.

Scenario 3: Reverse Drive - Fixed Target Distance, Different Speed (Result)

As for forward travel, different CAN messages were also sent for reverse travel, and the corresponding data are extracted from the ECU's memory address and represented below.

Memory Address for targetDistance: 0x70002f0c, read value Hex (Test case: one - 0x447A0000, two - 0x447A0000, three - 0x447A0000, four - 0x447A0000, five - 0x447A0000),
Memory Address for cumulativeDistance: 0x70002efc, read value Hex (Test case: one - 0x44878C12, two - 0x4484820B, three - 0x4487CC74, four - 0x44902C0A, five - 0x449F68B6).

Table 6.10: Test Result for Fixed Atomic Distance, Different Speed in Reverse.

Test Case	CAN Msg (Speed Duty)	Expected Speed in mm/s	Target Distance mm (Converted Float Value from Hex)	Traveled Distance in mm (Converted Float Value from Hex)	Test Result	Travel Deviation (mm)	Accuracy Obtained (%)
1	90	230	1000	1084.37	Passed	84.37	91.563
2	80	330	1000	1060.06	Passed	60.06	93.994
3	70	430	1000	1086.38	Passed	86.38	91.362
4	60	530	1000	1153.37	Passed	153.37	84.663
5	50	630	1000	1275.27	Passed	275.27	72.473

From the last result table, it is observed that, due to the increase in speed, the deviation is much higher, so the accuracy. It can be pointed out that the faster the car's travel speed, the more motion force will be added to the travel direction. Thus, accuracy for lower speeds will have the advantage over higher speeds. To analyze the speed change more, need to check the expected speed as per the guideline and the actual speed.

Scenario 4: Expected speed Vs. Actual Speed (Result)

Since it is already observed, an increase in the speed has a lower accuracy. So, in this result section, different speed data will be analyzed based on the speed mapping data used for algorithm implementation. “The speed mapping data was stated in the TUCminiCar documentation.” [52] The CAN messages were sent to test for three different speeds, and memory reading was performed 10 times each. After that, data were extracted from the ECU's memory address, which is represented below.

For the speed, there is a global variable declared, float32 globalCarSpeedValue
Memory Address for globalCarSpeedValue: 0x70002f04; read the value in Hex format.

110 Duty Cycle (230 mm/s):

Table 6.11: Memory Read for 110 Duty Cycle Speed.

Test Case	CAN Msg (Speed Duty)	Expected Speed in mm/s	Memory Read Value, Speed (Hex)	ActualSpeed in mm/s (Converted Float Value from Hex)	Speed Difference (mm/s)	Average Deviation
1	110	230	0x436BFF94	235.998352	5.998352	5.3805306
2	110	230	0x436EAE0E	238.6799	8.6799	
3	110	230	0x436AEC7B	234.923752	4.923752	
4	110	230	0x4368319C	232.193787	2.193787	
5	110	230	0x436B81FA	235.507721	5.507721	
6	110	230	0x436E51B1	238.3191	8.3191	
7	110	230	0x43678AD0	231.542236	1.542236	
8	110	230	0x436AF209	234.94545	4.94545	
9	110	230	0x436C56BA	236.338776	6.338776	
10	110	230	0x436B5B32	235.356232	5.356232	

150 Duty Cycle (630 mm/s):

Table 6.12: Memory Read for 150 Duty Cycle Speed.

Test Case	CAN Msg (Speed Duty)	Expected Speed in mm/s	Memory Read Value, Speed (Hex)	ActualSpeed in mm/s (Converted Float Value from Hex)	Speed Difference (mm/s)	Average Deviation
1	150	630	0x441F19B9	636.4019	6.4019	8.466777
2	150	630	0x441D2B78	628.6792	1.3208	
3	150	630	0x441CF942	627.8947	2.105347	
4	150	630	0x44222CCB	648.6999	18.6999	
5	150	630	0x441AD84F	619.3798	10.6202	
6	150	630	0x441DB8D6	630.8881	0.888062	
7	150	630	0x441A4674	617.1008	12.8992	
8	150	630	0x441DB296	630.7904	0.7904	
9	150	630	0x442184ED	646.077	16.07697	
10	150	630	0x4419C8A4	615.135	14.865	

200 Duty Cycle (1130 mm/s):

Table 6.13: Memory Read for 200 Duty Cycle Speed.

Test Case	CAN Msg (Speed Duty)	Expected Speed in mm/s	Memory Read Value, Speed (Hex)	Actual Speed in mm/s (Converted Float Value from Hex)	Speed Difference (mm/s)	Average Deviation
1	200	1130	0x4488657A	1091.17114	38.82886	21.562415
2	200	1130	0x448986CC	1100.2124	29.7876	
3	200	1130	0x448997BB	1100.74158	29.25842	
4	200	1130	0x4489865D	1100.19885	29.80115	
5	200	1130	0x4489EBEE	1103.3728	26.6272	
6	200	1130	0x448B1249	1112.57141	17.42859	
7	200	1130	0x448B42B3	1114.08435	15.91565	
8	200	1130	0x448C06EE	1120.21655	9.78345	
9	200	1130	0x448C26D3	1121.21326	8.78674	
10	200	1130	0x448C12FE	1120.59351	9.40649	

From the above three data tables, it is clear that the faster a car travels, the more fluctuating its actual speed. This fluctuation affects the distance calculation, and accuracy is more affected by the higher speed. In the next sub-chapter, an evaluation of the obtained results will be presented.

6.2 Evaluation

The evaluation chapter analyzes the results obtained during the whole system test. It is divided into two main sections: Performance Analysis (6.2.1) and Improvement Areas (6.2.2). These will be described in the following sections.

6.2.1 Performance Analysis

This section summarizes all the test results obtained during the whole project implementation. Since testing was done for the three main units, evaluation also needs to be wise.

- **Unit Test Performance Analysis**

In the unit test sections, all the logic implemented within the implementation was tested. In addition, input data acceptance was also checked for the developed application SWC. In this section, data were tested in terms of capabilities rather than accuracy. As all the test cases, including the simulation test, were successful, thus the unit test section can be declared 100% accurate.

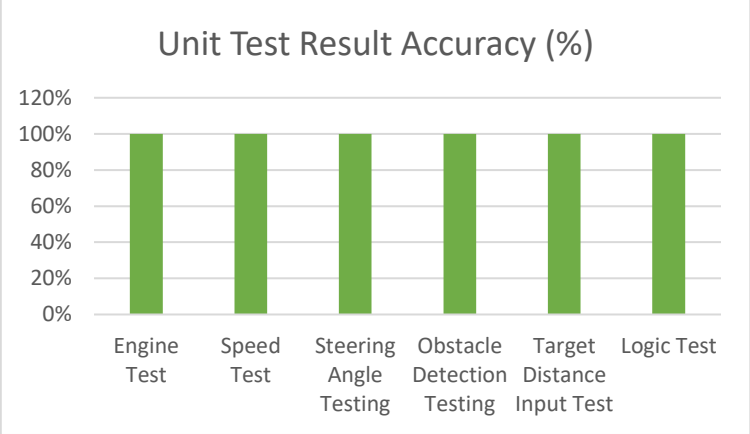

Figure 6.3: Unit Test Performance.

- **Integration Test Performance Analysis**

In the integration test section, the interaction between different SWCs and different components was tested. While all the SWC interactions were successful during testing, a few were less accurate in terms of achieving the target 100% accurately. All of those were related to traveling the target distance as per the distance needed to travel. There were a small number of deviations observed, for which it was not 100% accurate. The summary of the test section is presented in the following graph.

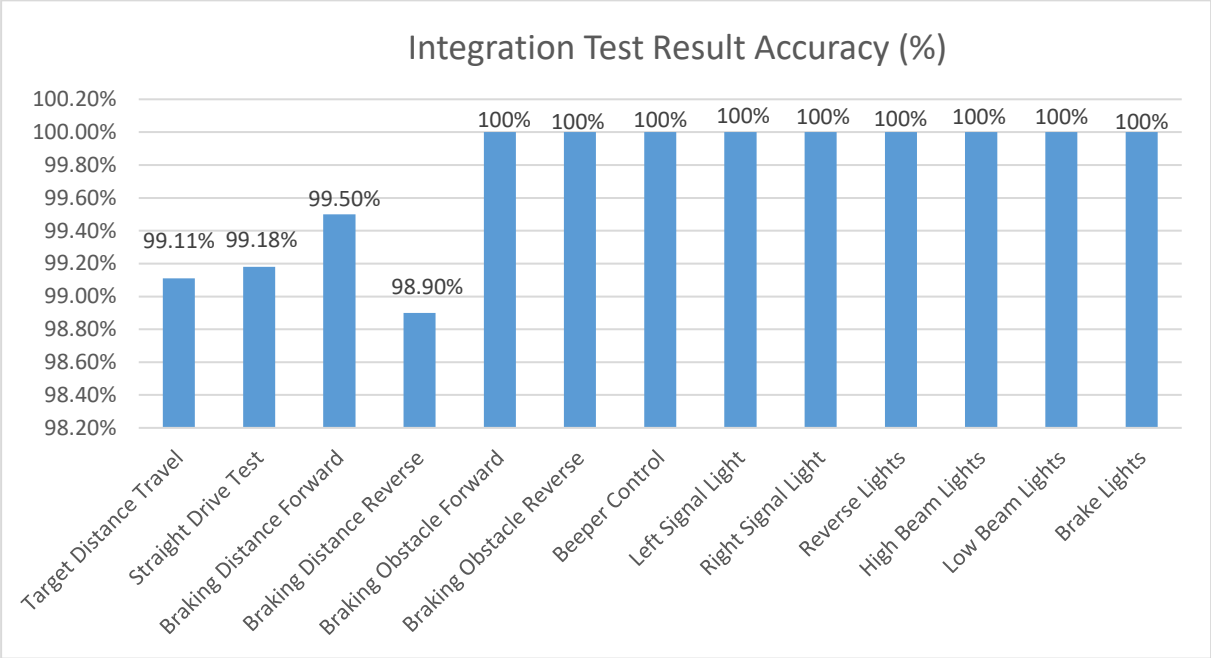

Figure 6.4: Integration Test Performance.

- **End-to-End (E2E) Test Performance Analysis**

The E2E test was conducted to check the car's overall straight-drive accuracy at different distances and speeds. After observing the different test cases, it was stated that the accuracy decreased when the target distance was reduced, and the parallel speed increased. In the graph below, the average percentage values for each criterion are presented for the five test cases.

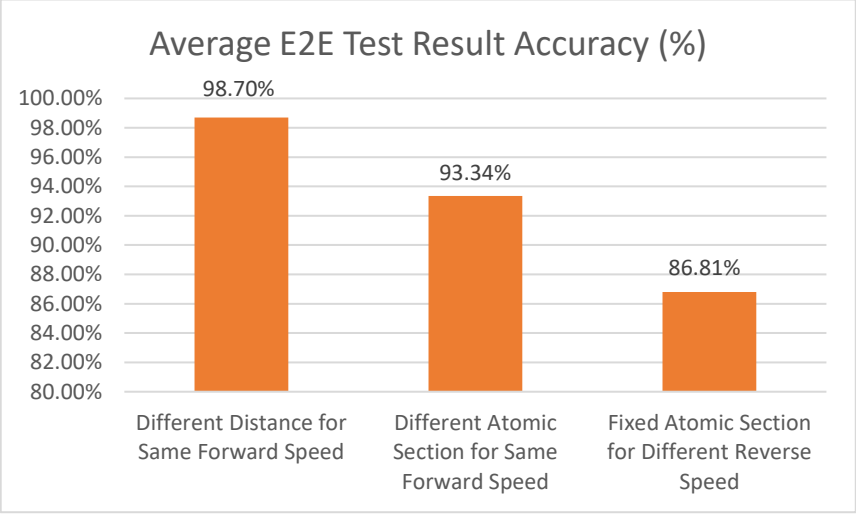

Figure 6.5: End-to-End (E2E) Test Performance.

It is cautiously observed that the more varying the speed, the less accurate the accuracy. So, to confirm this, another test was conducted on three different speed data sets (230mm/s equivalent to 110 duty cycles, 630mm/s equivalent to 150 duty cycles, and 1130mm/s equivalent to 200 duty cycles), reading at ten different times to check the status of speed fluctuations, which is represented below.

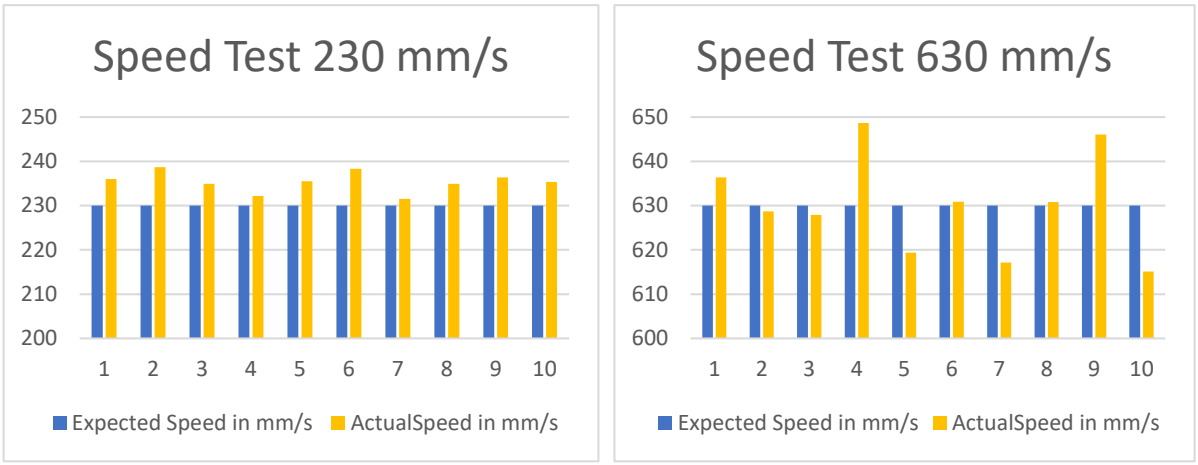

Figure 6.6: Speed Fluctuation Analysis for 230 mm/s and 630 mm/s.

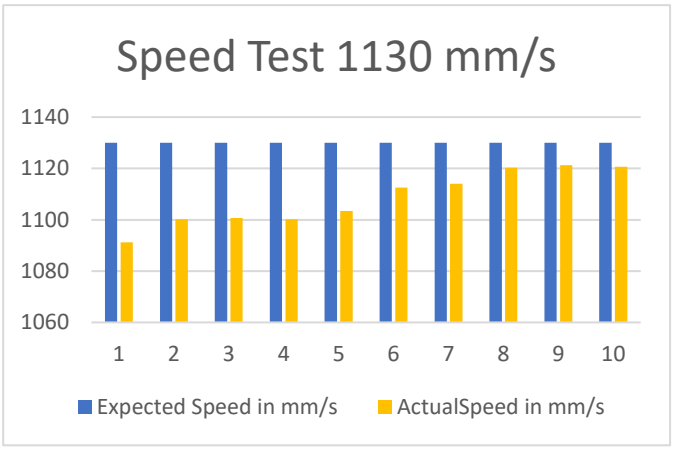

Figure 6.7: Speed Fluctuation Analysis for 1130 mm/s.

The three graphs mentioned above were constructed for ten different test cases, which are mentioned in the horizontal axes. After analyzing these graphs, it is clearly visible that the increase in speed fluctuates more in terms of actual vs. expected speed. Since speed data is counted every 10ms as per the designed runnable, this affects calculating distances since distance calculation depends on speed, which is already mentioned in the distance calculation logic in the chapter (5.1.3). In summary, it can be declared that, for less speed, the system is more stable than the higher speed, and the bottleneck is identified as the constant speed fluctuations.

6.2.2 Improvement Areas

As the bottleneck was already identified in the evaluation section, it is clearly visible that the problem remains at the hardware level. The DC motor, which is actually a racing car motor, fluctuates more in providing speed to the system for the higher speed range, especially in the reverse direction. It also has a mechanism to run every 20 ms, which is higher than the designed runnable, and a speed control mechanism, which is 10 ms. There is also some additional motion force for higher speed acceleration in natural. Since, within the current setup, distance calculation fully depends on the speed of the car, the accuracy differences cannot be minimized, especially for the higher speed range. These can be minimized if an installed motor provides an accurate constant speed value or a rotation sensor is installed. Then, the distance calculation mechanism can be modified to match the rotation count.

7 Conclusion

This chapter summarizes the whole research work. The thesis goal was to maintain an atomic straight driving pattern by implementing an AUTOSAR SWC. The goal was successfully and accurately obtained. The only bottleneck for the thesis was identified as fluctuation in speed, especially for higher speed. For this reason, automatic braking due to target distance coverage has slight deviation, especially when the path length is short and the speed is higher in the reverse direction. This chapter is divided into two sub-chapters, Conclusion (7.1) and Future Work (7.2), and described in the following.

7.1 Conclusion

This thesis identified the challenges of implementing an atomic straight drive pattern to an Advanced Driver Assistance System (ADAS) demonstrator within the standard of the AUTOSAR framework since the demonstrator was an RC car, which is a replica of real-world cars. The research successfully achieved its objective of designing, implementing, and validating an AUTOSAR software component that ensures real-time control and high precision for atomic straight driving. The key outcomes of the thesis are mentioned below:

Analyzing AUTOSAR in the Demonstrator: The AUTOSAR architecture within the demonstrator TUCminiCar was extensively analyzed. Knowing the whole structure is mandatory before developing an SWC within an existing architecture. The methodology chapter describes the analyzed architecture overview in detail.

Development Model: The necessity of using a development model was disclosed for the thesis implementation. The V-Model development process was described and followed throughout the whole research for designing, implementation, and testing procedures.

Designing AUTOSAR SWC: An application SWC component was created within the existing dSPACE SystemDescription file, and then the necessary implementation was also conducted within the file. After creating the application SWC, a simulation was performed to test its functionality and logic validation. The SIL test was successful, allowing the SWC to carry on the system integration.

Integration and Validation: Integration took place in the EB Tresos Studio, with the modified *.arxml file generated from the dSPACE. After the integration process was completed, the system was validated to generate the project. From the newly

generated project, new *.hex file was flashed into the ECU to prepare the test environment. During the entire integration and testing time, the AUTOSAR toolchain was followed for the development procedure.

Communication Protocol: CAN communication was used to communicate from the tester to the ECU. During the setup process, extensive ideas for working with the CAN protocol were gathered.

Precision and Accuracy: The atomic straight diving pattern with the developed SWC showed precision control and higher accuracy both for driving straight and automatic braking. An issue at the hardware level has been identified, which is affecting accuracy for the higher speed range.

Real-time Responsiveness: For different actuators, including auditory and visual, real-time response was observed, which is essential for the automotive domain.

In summary, the research was fruitful because it produced a fundamental component for ADAS functionality that can be used on a larger scale. Since the straight-driving functionality was developed for the smallest section of the car, it is easier to use on a bigger scale. In addition, obstacle detection was also implemented and tested, which provided other functionalities of ADAS. An automatic braking mechanism was established for both target distance coverage and obstacle detection, which provided more ideas for the driverless car's implementation process. The thesis's motivation was to move towards driverless vehicles, as human errors are the main cause of numerous accidents all over the world. So, in a single word, the thesis topic's "AUTOSAR Software Component for Atomic Straight Driving Patterns" main goal has been successfully achieved.

7.2 Future Work

There is always room for improvement in any work. The current straight drive functionality fully depends on true ground-level position since there is no mechanism to track the lane. It would be a great idea to have a grayscale sensor to track the lane, which would make straight driving more reliable. There is a great scope to work with the security mechanism, especially for controlling cars with CAN messages, as anyone can access the car and control it. To avoid this, any user verification can prevent taking control of the car.

For now, the thesis research has been completed successfully by considering every aspect of current availability and compatibility.

Bibliography

[1] “AUTOSAR Classic Platform.” Accessed: Jun. 23, 2024. [Online]. Available: https://www.autosar.org/standards/classic-platform

[2] V. T. Popović, M. Vulić, A. Davidović, and I. Kaštelan, “Modeling and Development of AUTOSAR Software Components,” *2019 IEEE 23rd Int. Symp. Consum. Technol. ISCT 2019*, pp. 313–316, Jun. 2019, doi: 10.1109/ISCE.2019.8901035.

[3] P. Barry and P. Crowley, “Embedded Linux,” *Mod. Embed. Comput.*, pp. 227–268, 2012, doi: 10.1016/B978-0-12-391490-3.00008-4.

[4] “Despite notable progress, road safety remains urgent global issue.” Accessed: Nov. 01, 2024. [Online]. Available: https://www.who.int/news/item/13-12-2023-despite-notable-progress-road-safety-remains-urgent-global-issue

[5] “Autonomous Vehicle Market Size to Hit USD 2,752.80 BN by 2033.” Accessed: Nov. 01, 2024. [Online]. Available: https://www.precedenceresearch.com/autonomous-vehicle-market

[6] T. Winkle, “Safety benefits of automated vehicles: Extended findings from accident research for development, validation and testing,” *Auton. Driv. Tech. Leg. Soc. Asp.*, pp. 335–364, Jan. 2016, doi: 10.1007/978-3-662-48847-8_17.

[7] “Car Accidents Are Caused by Human Error | Morris, King & Hodge.” Accessed: Nov. 02, 2024. [Online]. Available: https://www.mkhlawyers.com/blog/what-percentage-of-car-accidents-is-caused-by-human-error/

[8] “Autonomous Vehicle Market Size, Share, Trends | Report [2030].” Accessed: Nov. 03, 2024. [Online]. Available: https://www.fortunebusinessinsights.com/autonomous-vehicle-market-109045

[9] B. Huang, H. Dong, D. Wang, and G. Zhao, “Basic concepts on AUTOSAR development,” *2010 Int. Conf. Intell. Comput. Technol. Autom. ICICTA 2010*, vol. 1, pp. 871–873, 2010, doi: 10.1109/ICICTA.2010.571.

[10] “Layered Software Architecture.” Accessed: Nov. 03, 2024. [Online]. Available: https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf

[11] A. R. Paul and M. George, “Brushless DC motor control using digital PWM techniques,” *2011 - Int. Conf. Signal Process. Commun. Comput. Netw. Technol. ICSCCN-2011*, pp. 733–738, 2011, doi: 10.1109/ICSCCN.2011.6024647.

[12] “KIT_A2G_TC387_3V3_TFT - Infineon Technologies.” Accessed: Nov. 10, 2024. [Online]. Available: https://www.infineon.com/cms/en/product/evaluation-boards/kit_a2g_tc387_3v3_tft/

[13] M. E. Rahmani, A. Amine, and R. M. Hamou, “Sonar Data Classification Using a New Algorithm Inspired from Black Holes Phenomenon,” *Int. J. Inf. Retr. Res.*,

vol. 8, no. 2, pp. 25–39, Feb. 2018, doi: 10.4018/IJIRR.2018040102.

[14] M. A. Butt *et al.*, “Micro-electromechanical system based optimized steering angle estimation mechanism for customized self-driving vehicles,” *Meas. Control (United Kingdom)*, vol. 54, no. 3–4, pp. 429–438, Mar. 2021, doi: 10.1177/00202940211000076.

[15] “SystemDesk - dSPACE.” Accessed: Nov. 12, 2024. [Online]. Available: https://www.dspace.com/en/pub/home/products/sw/system_architecture_software/systemdesk.cfm

[16] “Classic AUTOSAR tooling EB tresos Studio – Elektrobit.” Accessed: Nov. 12, 2024. [Online]. Available: https://www.elektrobit.com/products/ecu/eb-tresos/studio/

[17] “TASKING - Infineon Technologies.” Accessed: Nov. 12, 2024. [Online]. Available: https://www.infineon.com/cms/en/tools/aurix-tools/Compilers/TASKING/

[18] “TASKING-Compiler Qualification Kit.” Accessed: Nov. 12, 2024. [Online]. Available: https://resources.tasking.com/sites/default/files/2021-03/TASKING-Compiler Qualification Kit_WEB.pdf

[19] “KIT_DAP_MINIWIGGLER_USB | Empower Your Debugging and Flash Programming with miniWiggler - Infineon’s Future-Ready Solution for High Performance Debugging and Flash Programming - Infineon Technologies.” Accessed: Nov. 13, 2024. [Online]. Available: https://www.infineon.com/cms/en/product/evaluation-boards/kit_dap_miniwiggler_usb/

[20] “| MHS Online-Shop.” Accessed: Nov. 13, 2024. [Online]. Available: https://www.mhs-elektronik.de/index.php?module=artikel&action=artikel&id=3

[21] “Dynamic Architectural Simulation Model of YellowCar in MATLAB/Simulink Using AUTOSAR System.” Accessed: Nov. 16, 2024. [Online]. Available: https://monarch.qucosa.de/api/qucosa%3A20580/attachment/ATT-0/

[22] J. Park and B. W. Choi, “Design and implementation procedure for an advanced driver assistance system based on an open source AUTOSAR,” *Electron.*, vol. 8, no. 9, Sep. 2019, doi: 10.3390/ELECTRONICS8091025.

[23] “Adaptive User Interface for Automotive Demonstrator.” Accessed: Dec. 02, 2024. [Online]. Available: https://monarch.qucosa.de/api/qucosa%3A78220/attachment/ATT-0/

[24] “Technische Informatik | Fakultät für Informatik | TU Chemnitz.” Accessed: Nov. 16, 2024. [Online]. Available: https://www.tu-chemnitz.de/informatik/ce/research/yellowcar.php

[25] B. Liu, H. Zhang, and S. Zhu, “An incremental V-model process for automotive development,” *Proc. - Asia-Pacific Softw. Eng. Conf. APSEC*, vol. 0, pp. 225–232, Jul. 2016, doi: 10.1109/APSEC.2016.040.

[26] “Validation and Verification for System Development.” Accessed: Nov. 17, 2024. [Online]. Available: https://de.mathworks.com/help/ecoder/gs/v-model-for-system-development.html

[27] S. Saroja and S. Haseena, “Functional and Non-Functional Requirements in Agile Software Development,” *Agil. Softw. Dev. Trends, Challenges Appl.*, pp. 71–86, Jan. 2023, doi: 10.1002/9781119896838.CH5.

[28] “Requirements on Runtime Environment.” Accessed: Nov. 20, 2024. [Online]. Available: https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_SRS_RTE.pdf

[29] “Specification of RTE Software.” Accessed: Nov. 20, 2024. [Online]. Available: https://www.autosar.org/fileadmin/standards/R23-11/CP/AUTOSAR_CP_SWS_RTE.pdf

[30] “Specification of ADC Driver.” Accessed: Nov. 21, 2024. [Online]. Available: https://www.autosar.org/fileadmin/standards/R23-11/CP/AUTOSAR_CP_SWS_ADCDriver.pdf

[31] “Specification of DIO Driver.” Accessed: Nov. 21, 2024. [Online]. Available: https://www.autosar.org/fileadmin/standards/R23-11/CP/AUTOSAR_CP_SWS_DIODriver.pdf

[32] “Specification of ICU Driver.” Accessed: Nov. 21, 2024. [Online]. Available: https://www.autosar.org/fileadmin/standards/R23-11/CP/AUTOSAR_CP_SWS_ICUDriver.pdf

[33] “Specification of CAN Driver.” Accessed: Nov. 21, 2024. [Online]. Available: https://www.autosar.org/fileadmin/standards/R23-11/CP/AUTOSAR_CP_SWS_CANDriver.pdf

[34] “Specification of MCU Driver.” Accessed: Nov. 21, 2024. [Online]. Available: https://www.autosar.org/fileadmin/standards/R23-11/CP/AUTOSAR_CP_SWS_MCUDriver.pdf

[35] “Complex Driver design and integration guideline.” Accessed: Nov. 21, 2024. [Online]. Available: https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_EXP_CDDDesignAndIntegrationGuideline.pdf

[36] “Software Component Template.” Accessed: Nov. 22, 2024. [Online]. Available: https://www.autosar.org/fileadmin/standards/R20-11/CP/AUTOSAR_TPS_SoftwareComponentTemplate.pdf

[37] “ApplicationSwComponentType – automotive wiki.” Accessed: Nov. 23, 2024. [Online]. Available: https://automotive.wiki/index.php/ApplicationSwComponentType

[38] “SensorActuatorSwComponentType – automotive wiki.” Accessed: Nov. 23, 2024. [Online]. Available: https://automotive.wiki/index.php/SensorActuatorSwComponentType

[39] “CompositionSwComponentType – automotive wiki.” Accessed: Nov. 23, 2024. [Online]. Available: https://automotive.wiki/index.php/CompositionSwComponentType

[40] “Port – automotive wiki.” Accessed: Nov. 23, 2024. [Online]. Available: https://automotive.wiki/index.php/Port

[41] “Software Component Internal behavior – automotive wiki.” Accessed: Nov. 23, 2024. [Online]. Available: https://automotive.wiki/index.php/Software_Component_Internal_behavior

[42] C. Hanxing and T. Jun, “Research on the controller area network,” *Proc. - 2009 Int. Conf. Netw. Digit. Soc. ICNDS 2009*, vol. 2, pp. 251–254, 2009, doi: 10.1109/ICNDS.2009.142.

[43] “CAN Bus Explained - A Simple Intro [2024] – CSS Electronics.” Accessed: Nov. 23, 2024. [Online]. Available: https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial

[44] “BOSCH, CAN Specification, Version 2.0.” Accessed: Nov. 23, 2024. [Online]. Available: http://esd.cs.ucr.edu/webres/can20.pdf

[45] “Controller Area Network (CAN) Basics.” Accessed: Nov. 23, 2024. [Online]. Available: https://ww1.microchip.com/downloads/en/Appnotes/00713a.pdf

[46] “(PDF) Low-cost USB2.0 to CAN2.0 bridge design for Automotive Electronic Circuit.” Accessed: Nov. 23, 2024. [Online]. Available: https://www.researchgate.net/publication/210264476_Low-cost_USB20_to_CAN20_bridge_design_for_Automotive_Electronic_Circuit

[47] “Specification of CAN Interface.” Accessed: Nov. 24, 2024. [Online]. Available: https://www.autosar.org/fileadmin/standards/R21-11/CP/AUTOSAR_SWS_CANInterface.pdf

[48] “Specification of PDU Router.” Accessed: Nov. 24, 2024. [Online]. Available: https://www.autosar.org/fileadmin/fileadmin/standards/classic/2-0/AUTOSAR_SWS_PDU_Router.pdf

[49] “Specification of Communication.” Accessed: Nov. 24, 2024. [Online]. Available: https://www.autosar.org/fileadmin/standards/R21-11/CP/AUTOSAR_SWS_COM.pdf

[50] “Specification of CAN State Manager.” Accessed: Nov. 24, 2024. [Online]. Available: https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_SWS_CANStateManager.pdf

[51] “Specification of Communication Manager.” Accessed: Nov. 24, 2024. [Online]. Available: https://www.autosar.org/fileadmin/standards/R21-11/CP/AUTOSAR_SWS_COMManager.pdf

[52] M.Sc. H. Aljzaere and M. Sevil, “miniTUCar_BSW_v0.29 Project Documentation,” TU Chemnitz, Germany, 2024.

Chemnitzer Informatik-Berichte

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

CSR-21-01 Marco Stephan, Batbayar Battseren, Wolfram Hardt, UAV Flight using a Monocular Camera, März 2021, Chemnitz

CSR-21-02 Hasan Aljzaere, Owes Khan, Wolfram Hardt, Adaptive User Interface for Automotive Demonstrator, Juli 2021, Chemnitz

CSR-21-03 Chibundu Ogbonnia, René Bergelt, Wolfram Hardt, Embedded System Optimization of Radar Post-processing in an ARM CPU Core, Dezember 2021, Chemnitz

CSR-21-04 Julius Lochbaum, René Bergelt, Wolfram Hardt, Entwicklung und Bewertung von Algorithmen zur Umfeldmodellierung mithilfe von Radarsensoren im Automotive Umfeld, Dezember 2021, Chemnitz

CSR-22-01 Henrik Zant, Reda Harradi, Wolfram Hardt, Expert System-based Embedded Software Module and Ruleset for Adaptive Flight Missions, September 2022, Chemnitz

CSR-23-01 Stephan Lede, René Schmidt, Wolfram Hardt, Analyse des Ressourcenverbrauchs von Deep Learning Methoden zur Einschlagslokalisierung auf eingebetteten Systemen, Januar 2023, Chemnitz

CSR-23-02 André Böhle, René Schmidt, Wolfram Hardt, Schnittstelle zur Datenakquise von Daten des Lernmanagementsystems unter Berücksichtigung bestehender Datenschutzrichtlinien, Januar 2023, Chemnitz

CSR-23-03 Falk Zaumseil, Sabrina, Bräuer, Thomas L. Milani, Guido Brunnett, Gender Dissimilarities in Body Gait Kinematics at Different Speeds, März 2023, Chemnitz

CSR-23-04 Tom Uhlmann, Sabrina Bräuer, Falk Zaumseil, Guido Brunnett, A Novel Inexpensive Camera-based Photoelectric Barrier System for Accurate Flying Sprint Time Measurement, März 2023, Chemnitz

CSR-23-05 Samer Salamah, Guido Brunnett, Sabrina Bräuer, Tom Uhlmann, Oliver Rehren, Katharina Jahn, Thomas L. Milani, Güunter Daniel Rey, NaturalWalk: An Anatomy-based Synthesizer for Human Walking Motions, März 2023, Chemnitz

CSR-24-01 Seyhmus Akaslan, Ariane Heller, Wolfram Hardt, Hardware-Supported Test Environment Analysis for CAN Message Communication, Juni 2024, Chemnitz

Chemnitzer Informatik-Berichte

CSR-24-02 S. M. Rizwanur Rahman, Wolfram Hardt, Image Classification for Drone Propeller Inspection using Deep Learning, August 2024, Chemnitz

CSR-24-03 Sebastian Pettke, Wolfram Hardt, Ariane Heller, Comparison of maximum weight clique algorithms, August 2024, Chemnitz

CSR-24-04 Md Shoriful Islam, Ummay Ubaida Shegupta, Wolfram Hardt, Design and Development of a Predictive Learning Analytics System, August 2024, Chemnitz

CSR-24-05 Sopuluchukwu Divine Obi, Ummay Ubaida Shegupta, Wolfram Hardt, Development of a Frontend for Agents in a Virtual Tutoring System, August 2024, Chemnitz

CSR-24-06 Saddaf Afrin Khan, Ummay Ubaida Shegupta, Wolfram Hardt, Design and Development of a Diagnostic Learning Analytics System, August 2024, Chemnitz

CSR-24-07 Túlio Gomes Pereira, Wolfram Hardt, Ariane Heller, Development of a Material Classification Model for Multispectral LiDAR Data, August 2024, Chemnitz

CSR-24-08 Sumanth Anugandula, Ummay Ubaida Shegupta, Wolfram Hardt, Design and Development of a Virtual Agent for Interactive Learning Scenarios, September 2024, Chemnitz

CSR-25-01 Md. Ali Awlad, Hasan Saadi Jaber Aljzaere, Wolfram Hardt, AUTOSAR Software Component for Atomic Straight Driving Patterns, März 2025, Chemnitz

Chemnitzer Informatik-Berichte

ISSN 0947-5125

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