



Universidade do Minho
Escola de Engenharia

**Safety Critical Middleware in
Communication Protocols**

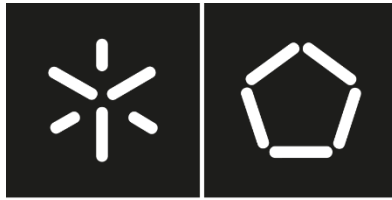
José Rui Amorim Gomes

José Rui Amorim Gomes

**Safety Critical Middleware in Communication
Protocols**

Uminho 2022

October 2022



Universidade do Minho

Escola de Engenharia

José Rui Amorim Gomes

**Safety Critical Middleware in Communication
Protocols**

Dissertação de Mestrado
Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Jorge Cabral

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do Repositório UM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho



**Atribuição
CC BY**

<https://creativecommons.org/licenses/by/4.0/>

Acknowledgements

Thanks, where thanks are due. Firstly, I would like to thank Professor Jorge Cabral for all the guidance and insights throughout the development of this work. I would also like to give my thanks to João Matos and Luís Vale for informing me about this opportunity in Bosch and encouraging me to pursue it.

A huge thanks to Jana Seidel for giving me an opportunity to prove myself that I could provide to a professional team relevant and quality work. For the guidance and advice, a huge thanks goes to Marco Esteves that accompanied me along the journey and helped me keep focused and creating good work.

To my colleague Emanuel Silva who joined Bosch with me and shared the same hardships and many hours of technical and casual conversations. Thank you for listening to me and the great friendship that we were able to create in that long year. To my Bosch colleagues João Santos, João Cardoso, Flávio Vasconcelos, Filipa Araújo and Margarida Almeida a huge thanks for the support and company along the year. Better that being in a great company is to be able to work in a great environment for which you all provided ample support and made me feel always welcomed and appreciative of the work I was doing. To Bosch as a company a thank you for showing me a corporate environment which is very professional and at the same time where one could find opportunities to express themselves.

To my family, for always being there for me every step of the way. To my mother, for always looking after me. To my dad, for keeping interested in everything I do and try to achieve. To grandparents, that sought to understand and showed genuine interest in the complicated terms I would always talk about whenever they asked about my progress. To my brothers, for being the greatest pain, but also the greatest companionship.

A special thanks to my good friend Sérgio Ferreira for being a very important influence all these years.

Thanks to all of those whom I was fortunate to cross paths with and made me the person I am today!

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

A área automóvel desde a sua criação está repleta de novas invenções e inovações tendo evoluído de sistemas puramente mecânicos para cada vez mais elétricos/eletrônicos (E/E). De entre as várias novas inovações surgiu o X-by-Wire cujo objetivo é melhorar o desempenho e segurança do veículo, num contexto onde as partes hidráulicas e mecânicas do sistema são parcialmente substituídas por sistemas elétricos e/ou eletrônicos [1].

Um sistema Steer-by-Wire substitui um sistema de direção convencional trocando a direção mecânica por uma completamente elétrica. É esperado por um destes sistemas dois serviços principais: o controlo da direção das rodas de acordo com o pedido do condutor e que proporcione uma força que emite o feedback que convencionalmente obtido quando se usa um volante tradicional [2].

Quando se opera neste tipo de sistemas “autónomos” é obrigatório que eles sejam fiáveis (fail-safe/fail-operational) sendo por norma classificados como sistemas de segurança crítica de controlo, onde um estado de falha num componente não pode levar a falha do sistema completo [2]. Um destes sistemas é o Steering Angle Sensor (SAS) que calcula a posição e velocidade do volante durante a sua operação e utilizando o protocolo CAN transmite esses dados para uma unidade central para posterior processamento e tomada de decisões.

Com esta dissertação implementou-se um driver de comunicação CAN (Controller Area Network) que suporta os protocolos CAN clássico e CAN-FD com flexibilidade quer nas suas configurações como nas mensagens que envia e recebe. Importante também que cumpra com as restrições elevadas de ambientes safe e tolerantes a falhas aproveitando as vantagens da escolha deste protocolo. Para isto, aplicou-se normas internacionais tais como a ISO 26262 (Functional Safety Standard) com os seus dois pontos chave: Segurança (Safety) e Qualidade Intrínseca [3] e outros guias como o MISRA. Complementando isto criou-se um GUI capaz de decifrar, analisar e simular mensagens CAN neste contexto.

Palavras-chave: Protocolo CAN; Drivers de dispositivos; Sistemas embebidos; ISO26262; Segurança Crítica;

Abstract

The automotive field was always filled with new inventions and innovations. It started by being completely mechanical and in recent years with successful iterations using many more electric and electronic subsystems. Among all the innovations, the X-by-Wire has the objective of improving the performance and safety of the vehicle, by replacing several mechanical and hydraulic parts of the system with their electronic or/and electric counterparts [1].

A Steer-by-Wire system replaces the conventional direction car system by eliminating the physical connection between the steering wheel and the wheels of a car by using electrically controlled motors to change the direction of the wheels [4]. It is expected for the system to guarantee two main services: the control of the wheels in accordance with the driver and that the steering wheel produces a feedback force that is usual from a traditional steering wheel [2].

In this type of “autonomous” systems is mandatory for them to be fail-safe, whereby normally are classified as critical safety control systems, defined by having a safe state where even if a component fails the whole system does not crumble to dust [2]. One of these systems is the Steering Angle Sensor, which calculates the position and velocity of the steering wheel during its operation and by using the CAN protocol transmits the calculated data to a central unit for further processing and decision making.

This dissertation has for its primary goal the implementation of a safe and flexible CAN communication driver with the objective of fulfilling the high restrictions of safe and fault-tolerant environments. Doing so by taking advantage of design standards and the CAN protocol. Standards such as the ISO 26262 (Road Vehicles – Functional Safety) focused on Safety and Intrinsic Quality and guidelines such as MISRA [3]. The driver must support both classical and CAN-FD configurations as well any type of messages. Adding to the driver, a GUI is also to be developed with display, analysis, and simulation capabilities for CAN messages.

Keywords: CAN Protocol; Device drivers; Embedded Systems; ISO26262; Safety-Critical;

Table of Contents

Resumo	v
Abstract	vi
Table of Contents	vii
List of Figures	x
List of Tables	xiii
Acronyms List	xiv
Introduction	1
1.1 Contextualization	2
1.2 Motivation.....	2
1.3 Objectives.....	3
1.4 Dissertation Structure	3
State of the Art	5
2.1 Embedded Systems	5
2.2 Critical Safety in Embedded Systems.....	6
2.2.1 Basic Concepts and Considerations	7
2.2.2 Design and Implementation	8
2.2.3 Test and Validation	8
2.2.4 ISO 26262 – Road Vehicle – Functional Safety	9
2.2.5 C programming language standards and guidelines	12
2.3 Communication protocols in Automotive.....	13
2.3.1 Communication principles in automotive.....	14
2.3.2 LIN Protocol	16

2.3.3 FlexRay Protocol.....	18
2.3.4 CAN Protocol.....	20
2.4 Conclusions.....	27
System Specification	28
3.1 Project Requirements.....	28
3.2 System Architecture.....	32
3.3 Hardware Specification	35
3.3.1 Microcontroller	35
3.3.2 CAN Module.....	40
3.3.3 CAN Transceiver.....	41
3.3.4 CAN Transceiverless.....	42
3.4 Conclusion	43
Implementation	44
4.1 Software Implementation	44
4.1.1 Development Environment.....	45
4.1.2 Software Guidelines.....	46
4.1.3 Software Layers.....	47
4.2 Software Workflow	49
4.2.1 State Machine	50
4.2.2 Initialization.....	51
4.2.3 CAN wake-up	52
4.2.4 Bus-off Event.....	55
4.2.5 Read and Write operations.....	56
4.3 CANoe Software.....	57
4.3.1 CANoe Environment	58
4.3.2 CANoe panels use cases	58

4.4 Conclusion	62
Tests and Results	63
5.1 Message system validation.....	63
5.2 Bus-off event validation	68
5.3 Non-transceiver validation	68
5.4 Standby operations validation.....	69
5.5 Panels validation.....	71
5.5.1 Demonstration panel	71
5.5.2 Configuration panel	72
5.5.3 Calibration panel	74
5.5.4 Simulation panel	75
Conclusion and Future Work	78
6.1 Conclusion	78
6.2 Future Work.....	79
Appendix A – Panels GUI	81
References	84

List of Figures

Figure 2-1: Overview of the ISO 26262 series standards [26]	11
Figure 2-2: Classic V Model [32]	12
Figure 2-3: LIN message frame.....	17
Figure 2-4: FlexRay data frame [41]	19
Figure 2-5: Conventional Networking vs CAN Bus Networking [50].....	21
Figure 2-6: Data rate relation with the bus line length	22
Figure 2-7: CAN Classic data frame	23
Figure 2-8: CAN remote frame	24
Figure 2-9: CAN Error frame	24
Figure 2-10: CAN-FD data frame.....	25
Figure 2-11: Steering wheels different implementations. With early implementations of a purely manual steering wheel (5a) and future implementations using hydraulic parts (5b) and electronic motors(5c)	26
Figure 2-12: Steer-by-Wire Steering.....	27
Figure 3-1: Representation of the relations between the system, software and hardware	32
Figure 3-2: AUTOSAR Partial System Stack	33
Figure 3-3: System Stack.....	34
Figure 3-4: S32K116 development board.....	37
Figure 3-5: S32K2TV development board.....	40
Figure 3-6: CAN transceiver	41
Figure 3-7: Transceiver with Wake-Up	42
Figure 3-8: Transceiver with Galvanic Isolation	42
Figure 3-9: Connection of CAN nodes without CAN transceiver [58]	43
Figure 4-1: S32 Design Studio (left) and MISRA (right) logos.....	45
Figure 4-2: CANoe logo.....	46
Figure 4-3: Project system stack folders	47

Figure 4-4: CAN Driver State Machine	50
Figure 4-5: Initialization Flowchart.....	52
Figure 4-6: The wake-up process begins with configuration. There are two defined processes for wake-up via pattern (6a) which requires no extra configuration and wake-up via frame (6b)....	53
Figure 4-7: The standby process differs within the family S32Kxxx because of the number of cores present in which one and the level of security enable in each one. Still the common ground starts by disable all the peripherals and start application shutdown.....	54
Figure 4-8: Flowcharts describing the wake-up processes in order to leave standby, via wake-up frame and preconfigured messages (8a) and via wake-up pattern depicted in the ISO 11898:2-2016 (8b)	55
Figure 4-9: Recover from Bus-Off flowchart	56
Figure 4-10: CAN main operations are the writing operation where there is the transmission of data to the CAN bus (Figure 4-10a) and the complementing operation of reading throw the reception of frames (Figure 4-10b).....	57
Figure 4-11: CANoe Environment.....	58
Figure 4-12: Use case demonstration panel	59
Figure 4-13: Use case simulation panel	60
Figure 4-14: Use case calibration panel	61
Figure 4-15: Use case configuration panel	62
Figure 5-1: Test scenario for the S32K116.....	64
Figure 5-2: Communication between the CANoe and the microcontroller	64
Figure 5-3 - Messages being transmitted at 2Mbps in the data phase and 500 kbps in the arbitration phase	66
Figure 5-4 - Graphical representation of the average bit rate with different baud rates	67
Figure 5-5 - Transmission time for 8 Mbps baud rate and 500 kbps on a 8 byte payload and 16 byte payload respectively	67
Figure 5-6 - Bus-off event and subsequently recover from the bus-off.....	68
Figure 5-7 - No transceiver measure and differential CAN bus measure	69
Figure 5-8 - Message mirroring the message sent on the no transceiver communication	69
Figure 5-9 - Demonstration Panel and the analysed message	71
Figure 5-10 - Demonstration panel status and error indication	72
Figure 5-11 - Reconfiguration of the CAN bus line	73

Figure 5-12 - Custom message to receive data from a new message	74
Figure 5-13 - Command requesting the software version	74
Figure 5-14 - Calibration and Decalibration of the sensor and its standard message.....	75
Figure 5-15 - Different stages of simulation	77

List of Tables

Table 2-1 - Automotive subsystems and influence of principal requirements 16

Table 2-2: Types of messages on LIN protocol 18

Table 3-1: Stakeholder Requirements29

Table 3-2: System Requirements 31

Table 3-3: Differences between CAN modules41

Table 5-1 - Latency of messages at different baud rates 65

Table 5-2 - Average data rates for different CAN velocities67

Table 5-3 - Test scenario testing the standby operation70

Acronyms List

ABS Anti-lock Braking System.

ACK Acknowledged.

ADC Analog to Digital Converter.

ADAS Advanced Drivers Assistant Systems.

ADR Adaptive Data Rate.

ASIL Automotive Safety Integrity Level

API Application Programming Interface.

AUTOSAR AUTomotive Open System ARchitecture.

CAN Controlled Area Network.

CAN-FD Controlled Area Network with Flexible Data.

CAPL Communication Access Programming Language

CPU Central Processing Unit.

CRC Cyclic Redundancy Check.

DLC Data Length Code.

DSP Digital Signal Processor.

DYN Dynamic Segment

ECU Electronic Control Unit.

ESP Electronic Stability Program

FTA Fault Tolerant Analysis.

GND Ground.

GPIO General Purpose Input Output.

GUI Graphical User Interface.

HAL Hardware Abstraction Layer.

I²C Inter-Integrated Circuit.

IDE Integrated Development Environment.

ISO International Standards Organization.

ISR Interrupt Service Routine.

IP Intellectual Property.

LIN Local Interconnect Network.

MCU Microcontroller Unit.

MISO Master In Slave Out.

MISRA Motor Industry Software Reliability Association.

MOSI Master Out Slave In.

MOST Media Oriented Systems Transport.

NACK Not Acknowledged.

NIT Network Idle Time.

OEM Original Equipment Manufacturer

PIT Programmable Interrupt Timer.

RS-232 Recommended Standard 232.

RTOS Real Time Operating System.

SAS Steering Angle Sensor

SBF Synch Break Field.

SbW Steer-by-Wire

SCK Serial Clock.

SD Secure Digital.

SoC System on Chip.

SPI Serial Peripheral Interface.

SS Slave Select.

ST Static Segment.

SW Symbolic Window.

SWC Software Component.

SWD Serial Wire Debug.

UART Universal Asynchronous Receiver Transmitter.

USART Universal Synchronous Asynchronous Receiver Transmitter.

USB Universal Serial Bus.

TTL Transistor-Transistor Logic

Chapter 1

Introduction

The automotive industry has been subjected to an exponential evolution in technology nowadays. Most of the advances seen in vehicles are the result of better electronics (hardware and software), which consequently increase the complexity of those systems [5]. Where recent cars may contain up to 80 ECUs (Electronic Control Units) and hundreds of Megabytes in Software [6]. All this complexity is not free of faults, which can arise from many reasons, from natural causes to mistakes made in the assembly process, for example. Eliminating all the events that could lead a system to a failure state is at very least challenging if not impossible. However, with the usage of fault-tolerant approaches, the system can be compliant with safety requirements. But how can safety be measured? A safe system is one that must not harm people, not even put them in dangerous situations. This hypothetic situation, often called residual risk, can be measured in three factors [7]: Severity – The potential injury; Exposure – The probability of occurrence; Controllability – The ability of the system to avoid the specified harm. These are the pillars in which an ASIL (Automotive Safety Integrity Level) system must comply with to achieve the standards regulated [7]. An ASIL system has a classification from A to D, where D is the stringent one at the rigorous level.

This dissertation addresses the issue of fault-tolerant chip architectures for automotive applications in a multiprocessor environment with lockstep. In lockstep architecture, two processors execute the same code in a synchronized way, normally with one system clock cycle apart. The outputs produced by both cores feed a checker unit, allowing it to identify possible inconsistencies between them. If any of that is detected, then the system must recover itself to a safe state and report the error, complying with the requirements of reliability and safety. The mechanism studied since the end of the twentieth century for its safety achievements is conducting the interest of the industry, especially the automotive one, because of its growth and more restrict demands on safety.

1.1 Contextualization

Since its creation, the automotive field has been barraged with new inventions and innovations. This led to improvements in the overall comfort and efficiency of vehicles all around. However these commodities come at a price of complexity, having increased the number of ECUs exponentially as well as the amount of data traded between them.

The amount of data drove improvements on different communication protocols to accommodate the larger loads. However not all data needs to be treated the same way as protocols such as MOST address the transfer of video and sound data across the different dedicated actuators, for example display screens and sound speakers. The requirements for automotive communications depend on the subsystems. However they can be summarized by fault tolerance, determinism, bandwidth, flexibility and security [8].

To improve on those requirements protocols such as CAN were forced to improve, resulting in a new improved version the CAN-FD. This new version increases the payload supported, the maximum data rates, enhanced reliability with improvements on the cyclic redundancy check and backward compatibility with CAN classic nodes in case it is required to mix both protocols.

With such improvements, a system with CAN-FD can handle more data at a faster rate increasing the number of applicable use cases, such as electric vehicles, ECU flashing, robotics, trucks, buses, secure CAN implementations and ADAS with safe driving. A good example of enhancing vehicle safety and performance is the implementation of X-by-wire systems. These systems require hazard studies and fail-operational system architecture following safety and reliability standards such as ISO26262.

1.2 Motivation

Safety is evermore present in everyday objects, it has become a necessity and mandatory for commercial systems. This pushes the markets to develop and implement systems and architectures fail-operational capable. Redundancy solutions and the rapid growth of the market, create an immeasurable amount of systems that require to trade information. Part of it is crucial information that requires safe and reliable communication protocols.

In the automotive field there are several used communication protocols, each with strong points and focused applications. Since the criticality of information for the radio is not the same as the speed and direction of the wheels to the central unit for example. Also the increased

number of ECU in a car has increased drastically, could even top 80 per car. All this information needs to be transferred and interpreted with different grades of safety.

This is why it is important when devising an automotive project to clearly set the communication protocol to be used and which design guidelines and standards to use. In order to create clearness and objectiveness in the software developed.

1.3 Objectives

After taking into consideration the motivation, the final product of the dissertation is the creation of critical safe middleware for a CAN driver to interface the user and the microcontroller, using safety guidelines for the code, supporting the driver there will be the development of a GUI to provide better testing capabilities and visualization of the capabilities of the working driver. Therefore, the main goals of this dissertation are as follows:

- Analysis of CAN protocol, as a safe protocol;
- Development of CAN driver functional in the S32Kxxx NXP's microcontroller family;
- Development of graphical interface capable of simulation, analysis, test and validation of CAN;
- Develop a user-friendly graphical user interface;
- Software implementation following guidelines from MISRA and ISO 26262 standards to achieve a safe system;
- Analyse and validation of data on the CAN interface;
- Tests and validation of the CAN driver.

1.4 Dissertation Structure

The following document is split into six chapters, it has a top-down approach where its structure follows a logical order according to the development process that occurred during this Master's Thesis.

The first chapter introduces the current technological concepts, referring to the context and the motivation for the development of this project, as well as its objectives.

The second chapter focus on all basic concepts integrated into this project or that influenced it in different ways. Describing a more in-depth approach of developing critical safe software as well as associated standards. It also grasps concepts on different communication protocols with their diverse pros and negatives. Diving deeper into the used protocol of CAN and its close variant, CAN-FD.

The third chapter gives an overview of the system, and a further selection of which components were chosen and the reasoning for those choices.

The fourth chapter is divided into two sections corresponding to the hardware and software implementations. It focuses on how this project was developed, and explains the path taken.

Chapter five shows the tests that were made, along with some considerations about the obtained results.

Chapter six presents the main conclusions relative to this project, as well as future improvements that can be made.

Chapter 2

State of the Art

This master's dissertation is framed into the scope of safety in communication in Embedded Systems, and as of such in this chapter there will be explanations about the essential concepts about this theme.

The first section deals with concepts regarding embedded systems description. In the section 2.2, Critical Safety in Embedded Systems is addressed and explained in the context of the master's thesis, where important notions such as the ISOs (Standards) are explained. In the final section concepts and explanations about communication protocols will be addressed, as well as its implications, and which situations are applicable.

In this chapter, there is a collection of technological views and discussions on the topics previously mentioned varying in relevance for this dissertation.

2.1 Embedded Systems

Embedded systems cover a wide range of systems in our current world, as they crept into pretty much every type of device designed by mankind. Even so it is acknowledged that an embedded system is, in contrary to a general-purpose system, a system dedicated to performing a small number of tasks, varying in complexity [9]. It can also be defined citing Michael Barr, as "A combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function. In some cases, embedded systems are part of a larger system or product" [10]. This means that the personnel computers are excluded from being considered an embedded system, but even with the number of new CPUs growing per year, around ninety-eight percent of all microprocessors manufactured are used in embedded systems [11]. Its fast growth can be explained by several main reasons: they function as a replacement for discrete logic-based circuits, provide functional upgrades, provide easy maintenance upgrades, improves mechanical performance and others [12].

The case is that the need for embedded systems is growing every year as they have already been applied in numerous fields, such as industrial control systems, information appliances,

communication equipment, medical instruments, intelligent instruments, and many others. As processors become more powerful, electronic products become more complex, including graphical user interfaces, communication networks, and databases [13]. Practical examples of such are microwave ovens, cell phones, calculators, digital watches, etc. [10].

At the architectural level, an embedded system represents the interaction between hardware and software elements, whose details maintained hidden in a way to have only information about the behavioural and relational levels. These elements can be internally implemented in the embedded system device or externally implemented interacting with the internal elements as well as with the external environment [14].

2.2 Critical Safety in Embedded Systems

As electronic control systems increase in both complexity and control authority there must be a commensurate increase in our ability to design and implement these systems safely. But a completely "safe" software cannot be guaranteed, within a reasonable engineering effort [15].

The explosive growth in microprocessor technology as well as ever-increasing application demands, has stimulated customers to demand increasingly higher levels of functionality and performance in the systems they procure. The large or more complex design solutions, which are delivered as a result pose complex safety and operational problems, and which are not addressable by conventional "hardwired logic" approaches, as in the past. Where safety provision is addressed purely in hardware, ALL potential hazards need to be identified in advance of the actual event, and suitably allowed for in the system design (not the easiest of tasks in large or complex systems). Also, the "law of diminishing returns" will apply in such hardware respects, in that the more safety provision that is made in hardware, the greater the potential for the protective mechanisms themselves to fail with time (and in turn the more backup circuits that will need to be provisioned, etc.) [15].

Microprocessor-based control systems are now commonplace in vehicles. These systems usually have a well-defined safe state. Many manufacturers and component suppliers are now experimenting with systems whose failure can have much more serious consequences such as: control by wire and supervisory systems, which can override the driver's inputs: and complex inter-connected systems where one failure can affect several others.

The safety record of automotive electronic systems has been very good, but to maintain this situation, the safety and reliability attained by these systems must increase at a rate equal to their complexity [16].

As electronic control systems increase in both complexity and control authority there must be a commensurate increase in our ability to design and implement these systems safely. The use of common specifications, hardware and software are all seen as potentially hazardous for very high integrity systems. Diversity is regarded as the best approach for providing the safety levels required and the preferred approach is to provide this by utilizing checking and back-up systems that are designed against a different requirement and have less functionality than the main control system. In this way the integrity of these systems can be kept high at a reasonable cost [17].

2.2.1 Basic Concepts and Considerations

As stated before is not possible to create a totally safe software no matter what the type or nature of its development, as safety is as likely to be as much dependent upon the underlying hardware or interaction with humans as upon any particular provision in software [15]. First, it is important to define some concepts such as Safety Critical System [18] being a system in which any failure or design error has the potential to lead to injury or loss of life, loss or severe damage to equipment and/or property, extreme financial losses or serious environmental damage [19] [20].

Fault – An abnormal condition that occurs inside a system. These failures can be classified depending on their persistence, where they can be transient, if they occur in a brief space of time, or permanent, in the opposite situation. But software faults are always permanent, what can vary it is their reproducibility, being divided into solid (hard or bohrbugs [21]) or elusive (soft or heisenbugs [21]).

Error – It is defined by being a discrepancy between the value obtained by the system and the theoretical value previously calculated. An error is also a part of the system state that may cause a subsequent failure [22].

Failure – Results from the system having faults and errors, culminating in its inability to provide a certain pre-determined function [22]. There are at least 3 sources of failure [16]:

- **Random Failures:** of electrical components, connection, wiring and mechanical devices are amenable to statistical prediction and used to estimate hazard potential, even though that task is not simple and failure probabilities can be produced;

- **Systematic Errors:** can occur in the system requirements, the hardware design or the software design. They are not random, with the same initial conditions will result in the same incorrect behavior. It results from human errors in the designing process or the tools used in the process. The adoption of internationally agreed standards on classification systems (e.g. ISO 26262) in order to minimize these types of errors and liability claims;
- **Intermittent Failures:** failures that are hard to predict and depend on factors outside the suppliers' control, such as EMC (Electromagnetic Compatibility) or the environment that the vehicle operates in. In a similar strategy as before the use of internationally agreed test standard is advised.

In summary there is a tight relation between fault, error, and failure, where a fault is a defect, an error is a corrupted state, and a failure is an event to be avoided [22].

2.2.2 Design and Implementation

Developing software for monitoring, control, or display of safety-critical functions is both challenging and expensive. Software for when safety is taken into account is perfect for powerful diagnostics and error handling techniques to be applied during run time. However, this requires the software to be developed in a more disciplined and scientific manner. Using well specified design and development policy is a necessity when intending to create safety systems, from the language, operating systems, tools, platforms, targets and others.

Good development methods and practices at each and every stage of developing is required to guarantee the quality and reliability of the system. The key concerns in software design are uncontrolled complexity and undetected change from the expected result. Therefore, the development methods and processes should aim at these issues.

Both can be addressed by different analytical techniques and rigorous configuration management (underlays all that takes place through the development life cycle and extends to subsequent operational and maintenance phases) and control procedures [15].

2.2.3 Test and Validation

Validation is the process of ensuring that the requirements are correct and complete. (Are we building "the right house"?). Verification, on the other hand, is the process of ensuring that the implementation satisfies the requirements. (Are we building "the house right"?) Validation of

requirements can be accomplished by four processes: traceability, reviews and analysis, simulation, and experience. Validation of requirements is expensive, but it is also money well spent. Identifying and eliminating errors early in the life cycle is cheaper and reduces “risks” later in the life cycle. Verification consumes almost 20-50 percent of the resources during the development of safety-critical software. Suitable for system analysis are also proposed for software analysis e.g. software FTA (fault tree analysis) [23].

Formal methods are extremely valuable by allowing precise mathematical analysis of specifications and designs, enabling more rigorous examination and therefore certification. Besides, they are ready for automation of test sequences and data. However, these methods can be difficult to apply depending on the environment and difficult to understand. The extent of the testing and validation process may depend on the safety certification expected to be achieved, therefore developers to achieve successful results need to ensure a comprehensive metrics policy, appropriate for their environment [15].

2.2.4 ISO 26262 – Road Vehicle – Functional Safety

ISO 26262 is an international standard focused on the safety of automotive electrical/electronic systems (E/E systems). It is with that premise that the ISO guides the developed embedded systems for road vehicles (e.g., cars, motorbikes, trucks, and other vehicles under the weight of 3000Kg) to be designed with an appropriate level of rigor for their intended application.

To understand what an appropriate level of rigor entails, it is necessary to first analyze the two aspects of the system development that the ISO 26262 covers: Safety and Intrinsic Quality, to a lesser extent. Where Safety focuses on ensuring that failures in the system software do not lead to (external) conditions that could cause harm to people, Intrinsic Quality emphasizes 'good' design, its simplicity, robustness, maintainability, testability, and others. In a design with high Intrinsic Quality is expected for it to perform more safely, and therefore easier to be readily demonstrable to be safe. [3]

However, no system is completely safe, but this standard has guidelines with the purpose of avoiding and control failures, therefore reducing the risk of people getting injured and possibly dying. To do this it has three pillars upon which is evaluated the risk and dangerous events [12]:

- **Severity (S)** – it is defined by the seriousness of the damage to one’s life. In which S1 would be the lowest classification, with only light injuries, and S3 with be the highest and worst category.

- **Exposure (E)** – it determines the probability of exposure to a dangerous situation. And similarly, to the classification in severity, E1 would be the least probable exposure and the E4 would be the highest.
- **Controllability (C)** – it is defined by the degree of control the driver has over its automotive when there is a failure in the system, where having a C1 classification means losing only a small part of the control and C3 a big loss in the vehicle control.

Weighing the classifications above it is possible to classify the risk of the system into a level of ASIL. The Automotive Safety Integrity Level (ASIL) is a classification system defined by the ISO 26262 and evaluates the risk and dangerous events to a client [24]. It can be divided into four classifications from ASIL A, where safety requirements for the system are the most basic to ASIL D, where safety requirements are the most complex and severe. There is still a classification for hazards that are identified as QM which do not dictate any safety requirements [25].

The standard is an extensive document that is divided into 12 parts, where in Figure 2-1 shows an overview of the standard ISO 26262:

- **Part 1:** Vocabulary;
- **Part 2:** Management of functional safety;
- **Part 3:** Concept Phase;
- **Part 4:** Product development at the system level;
- **Part 5:** Product development at the hardware level;
- **Part 6:** Product development at the software level;
- **Part 7:** Production, operation, service and decommissioning;
- **Part 8:** Supporting Processes;
- **Part 9:** Automotive safety integrity level (ASIL) – oriented and safety-oriented analysis;
- **Part 10:** Guidelines on ISO 26262;
- **Part 11:** Guidelines on application of ISO 26262 on semiconductors;
- **Part 12:** Adaptation of ISO 26262 for motorcycles.

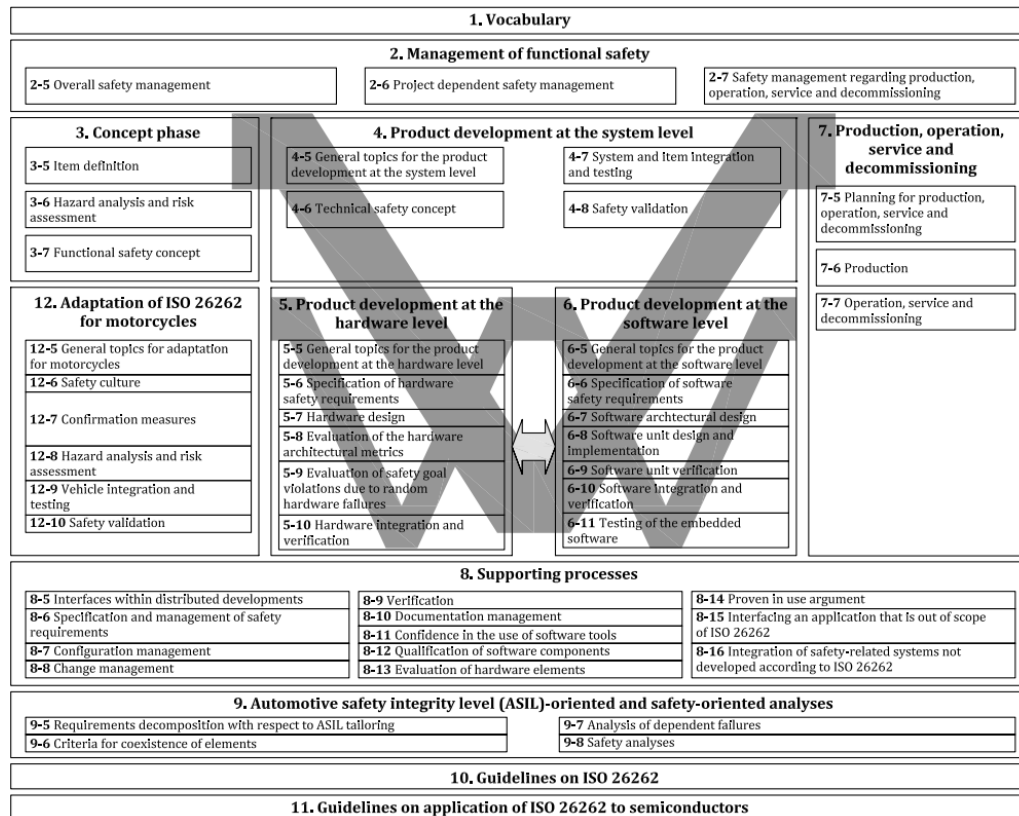


Figure 2-1: Overview of the ISO 26262 series standards [26]

From all these parts only some of them are relevant to the work in hands, where some take the job of being more informative and others are intended to serve as guidelines when developing the system and code. Part 1 defines the language of ISO 26262 its terms, abbreviations, acronyms, and others [27]. Part 2 is a guide focusing on the management of safety requirements, both from a project and organizational point of view [28].

Parts 4 and 6 are more directed for instructions during developing phases of the project, where part 4 is concerned with systems-level development, which entails detailed requirements analysis, system synthesis, functional and logical allocation, and system evaluation, validation and verification [7] and part 6 it reflects on general topics for product development at the software level and specification of the software safety requirements as well software architectural design, unit design and implementation, unit verification, integration and testing of the embedded software [29].

And finally, part 9 gives requirements and guidance concerning safety analyses and in particular, all aspects related to ASIL-oriented requirements [30]. And part 10 provides an overview of the ISO 26262 series of standards, as well as giving additional explanations, and is intended to enhance the understanding of the other parts of the ISO 26262 series of standards,

being of informative character only where its explanation expands from general concepts to specific concepts [31].

Since the ISO 26262 is not a process, it only applies additional constraints to the process already being implemented, focused on the system safety aspects using a classical V-model (standard way of describing the relationship between development artefacts) framework to organize its requirements, as shown in Figure 2-2.

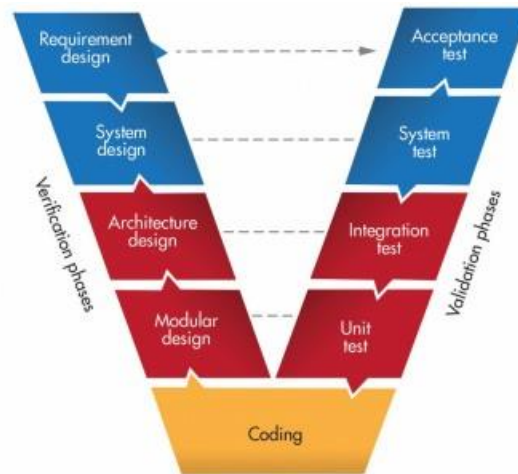


Figure 2-2: Classic V Model [32]

2.2.5 C programming language standards and guidelines

There are one million different ways to program the same snippet of code. However when working within a team all contributing to the whole project, there is a need to create certain rules and guidelines, so each piece of code can be read and understood by all the members. That is the reason to create documents such as the MISRA and coding rules by the AUTOSAR organization.

The MISRA guidelines for C were first created in the year of 1998. Providing coding standards for developing safety-critical systems. There are checking software capable of understanding if a code was written used such guidelines, this ensures that the code produced is safe, secure, reliable and portable for different compilers.

Nevertheless achieving MISRA compliance takes knowledge, skill, and the right tools [33]:

- **Know the rules:** You need to know the MISRA coding rules pertinent to which version of C or C++ you're using;
- **Check your code constantly:** Continuously inspecting your code for violations is the best way to improve quality.

- **Set baselines:** Embedded systems come with legacy codebases. By setting baselines, you can focus on making sure your new code is compliant.
- **Prioritize violations based on risk:** You could have hundreds or even thousands of violations in your code. That's why it's important to prioritize rule violations based on risk severity. Some static code analysis tools can do this for you.
- **Document your deviations:** Sometimes there are exceptions to the rule. But when it comes to compliance, every rule deviation needs to be well-documented.
- **Monitor your MISRA compliance:** Keep an eye on how MISRA compliant your code is. Using a static code analyser makes this easier by automatically generating a compliance report.
- **Choose the right static code analyser:** Choosing the right static code analyser makes everything else easy. It takes care of scanning your code — new and legacy — for violations. It prioritizes vulnerabilities based on risk.

The compliance with the MISRA goes deep into the development process and the development team skills. From the framework, training, style guide, metrics, tool management, compiler and its configuration, static analysis tools and their configuration and validation, and ending at the run-time behaviour. All these steps are required to be at some degree evaluated for a software to be compliant [34].

The MISRA C coding standard was originally written for the automotive industry. But today, MISRA standards for C and C++ are widely used by embedded industries — including aerospace, military defence, telecommunications, medical devices, and many others. Most of these industries have a compliance requirement to use a coding standard [33]. There are many other standards for programming in C, however, in the automotive field, there is a need to mention about the specification of C Implementation Rules, from the AUTOSAR organization, which aims to enhance software quality by avoiding the use of risky language constructs and ease portability to other compilers or microcontroller platforms [35].

2.3 Communication protocols in Automotive

A communication protocol is defined by being a system of rules that allow two or more entities to exchange information via any kind of variation of a physical quantity. The protocol defines the rules, syntax, semantics and synchronization of communication and possible error recovery

methods [36]. However, only some protocols are automotive ready where safety is of the highest importance, and that why protocols such as CAN, FlexRay, MOST, ethernet and LIN were born.

As mentioned above there are several automotive protocols and variations from the most established ones, which makes it impossible to cover every single one of them. Because of that only the most important ones and most similar in scope to the CAN protocol (which is the main protocol that will be implemented in this thesis) will be briefly introduced.

2.3.1 Communication principles in automotive

Integral functionalities such as driver assistance and autonomous driving and others created variable applications that demand information exchange between the ECUs, sensors and actuators. These applications require diverse qualities of service, being data transmission rates one of the most important [37]. Variable requirements in combination with large number of communication nodes and limited bus capacities led to a functional separation of bus segments into subsystems. Communication requirements addressed by different fieldbus technologies include [8]:

- **Fault-tolerance:** fault tolerant (normally safety-critical) communications are built to tolerate defective circuits, line failures and other types of failure. They do this by using redundant hardware and software architectures. Moreover, this type of communication should provide error containment.
- **Determinism:** deterministic communications guarantee timeliness, allowing to know exactly the transmission time for a message. This is required for safety critical automotive systems with strong real-time requirements. Examples of determinism are messages being sent within precise time intervals or at predefined time instants.
- **Bandwidth:** normally there is a trade-off between required bandwidth, the cost of providing such bandwidth and the level of subsystem integration possible with a single shared communication bus. However, recent automotive communication protocols provide high bandwidth allowing for the latest automotive subsystems working together with high degree system integration.
- **Flexibility:** the ability of a communication protocol has to handle, for example, event-triggered and time-triggered messages, the capacity to cope with varying loads and/or number of messages and accesses into the network, scalability and extensibility of a network.

- **Security:** guarantee the security of the system (i.e. no authorized accesses to the system are possible) in case of the communication is reachable from the outside of the automotive, especially in diagnostics tools, wireless connections and telematics.

In an automotive system there are several subsystems that rely on networking with different and defined jobs. Of all automotive subsystems there are eight typical types that can be distinguished [8]:

- **Chassis systems:** part of the vehicle active safety systems which require feedback control. Systems such as ESP, designed to assist the driver in over-steering, under-steering and roll-over situations, and ABS, designed to help the driver maintain steering capabilities and avoid skidding during breaking.
- **Air-bag systems:** part of vehicle passive safety systems responsible for the operation of the airbags in a vehicle. From the sensors that detect abnormal situations to the appropriate response depending on the type of situation.
- **Powertrain:** is the set of parts responsible for taking power from the engine of the vehicle to the driving axis, passing through the gear box. This engine control oversees coordination of fuel injection, engine speed, valve control, cam timing and others.
- **Body and comfort electronics:** these types of systems normally rely on driver interaction, are not safety-critical and require discrete control.
- **X-by-Wire:** subsystems that replace hydraulic and mechanical parts with electronics and computer (feedback) control systems. For example, steer-by-wire, shift-by-wire, throttle-by-wire, and break-by-wire.
- **Multimedia and infotainment:** systems include for example, car stereos, speakers, GPS, monitors, video games, voice processing, HMI, Internet connectivity etc.
- **Wireless and telematics:** intercommunication of wireless devices and telematics functions such as traffic information, fleet management systems, maintenance systems and anti-theft systems.
- **Diagnostics:** diagnosing of components and properties, service and maintenance with the possibility of downloading and updating software.

With an overview about the subsystems and the main requirements it is possible to map the level for each requirement to a subsystem, in the

Subsystem	Communication requirements				
	Fault-tolerance	Determinism	Bandwidth	Flexibility	Security
Chassis	Yes	Yes	Some	No	No
Airbag	Yes	Yes	Some	No	No
Powertrain	Some	Yes	Yes	Some	No
Body and Comfort	No	Some	Some	Yes	No
X-by-Wire	Yes	Yes	Some	No	No
Multimedia / Infotainment	No	Some	Yes	Yes	No
Wireless / Telematics	No	Some	Some	Yes	Yes
Diagnostics	No	Some	No	Yes	Yes

Table 2-1 - Automotive subsystems and influence of principal requirements

A segmented topology (divided by subsystems or partial subsystems) brings great advantages in the dependability of the communication for critical applications:

- Only a small number of components is accounted and interconnected by a single segment.
- Every single bus segment can be configured in a way that is exactly matching the specific application requirements.

On the other hand, the application functions of the vehicles become more complex and require information exchange across several bus segments. This leads to an additional load for the ECU's acting as gateways between bus segments [37].

2.3.2 LIN Protocol

The LIN protocol was created in 1990, by the LIN Consortium and counted with the participation of automotive manufacturers such as BMW, VW, Audi, Volvo Mercedes-Benz, Volcano Automotive and Motorola. Its last iteration and standardization occurred in 2016 with the release of ISO 17987:2016 [38].

LIN bus works as a supplement to CAN bus. It offers drastically lower costs at the expense of lower performance and reliability, where fault tolerance is not critical. It uses a master-slave implementation up to 16 slaves with a single 12V wire (plus ground) based on ISO 9141 (K-line) physical layer. Achieving maximum speeds of 20kbps and bus length distances up to 40 meters.

LIN protocol supports wake up, sleep, and time-triggered scheduling with guaranteed latency time operations with variable data length (1 to 8 bytes). For safety it includes error detection and checksums.

The LIN protocol has seen an increase in the number of nodes in the automotive field being sued in door systems, windshield wiper motors, rain sensors, headlight levelling motors, signal indicators and many others. It also started gaining traction in non-automotive applications, such as remote switch panels, washing machines, dryers, printers and many other applications and devices [39].

A LIN frame consists of a header and a response part. Only the master can to initiate communication by sending the header part of the frame. If the master wants to send data to the slave it also sends the response, otherwise the master is requesting data and the slave send the response part [40].

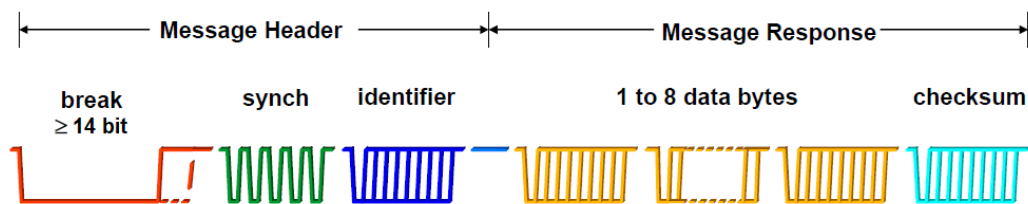


Figure 2-3: LIN message frame

On the Figure 2-3 it is possible to see the mentioned LIN message frame structure.

- **Synch Break Field (SBF)** - acts as a "start of frame" notice to all LIN nodes on the bus, is composed of synch break and the synch break delimiter.
- **Synch Field** - The 8 bit Sync field has a predefined value of 0x55 (in binary, 01010101). This structure allows the LIN nodes to determine the time between rising/falling edges and thus the baud rate used by the master node.
- **Identifier** - The Identifier is 6 bits, followed by 2 parity bits. The ID acts as an identifier for each LIN message sent and which nodes react to the header. Depending on the ID the slaves can react differently:
 - Ignore the subsequent data transmission.
 - Listen to the data transmitted from another node.
 - Publish data in response to the header.
- **Data** – when a slave is polled data is sent to the master and since LIN 2.0 the data length depends on the ID range (ID 0-31: 2 bytes, 32-47: 4 bytes, 48-63: 8 bytes).

- **Checksum** - checksum field ensures the validity of the LIN frame.

To conclude the LIN topic, it is required to talk a little about the six different types of frames that exist, as shown in the Table 2-2 below.







	Unconditional Frames	The default form of communication where the master sends a header, requesting information from a specific slave. The relevant slave reacts accordingly
	Event Trigger Frames	The master polls multiple slaves. A slave responds if its data has been updated, with its protected ID in the 1st data byte. If multiple respond, a collision occurs and the master defaults to unconditional frames
	Sporadic Frames	Only sent by the master if it knows a specific slave has updated data. The master "acts as a slave" and provides the response to its own header - letting it provide slave nodes with "dynamic" info
	Diagnostic Frames	Since LIN 2.0, IDs 60 and 61 are used for reading diagnostics from master or slaves. Frames always contain 8 data bytes. ID 60 is used for the master request, 61 for the slave response
	User Defined Frames	ID 62 is a user-defined frame which may contain any type of information
	Reserved Frames	Reserved frames have ID 63 and must not be used in LIN 2.0 conforming LIN networks

Table 2-2: Types of messages on LIN protocol

2.3.3 FlexRay Protocol

FlexRay is an automotive network communications protocol that was developed by the FlexRay Consortium (by BMW, Bosch, Daimler-Chrysler and Philips in 2000) [41] to govern on-board automotive computing [42]. It is designed to be faster and more reliable than CAN and TTP, but it is also more expensive. The FlexRay consortium disbanded in 2009, but the FlexRay standard is now a set of ISO standards, ISO 17458-1 to 17458-5. It supports communications up to 10Mbits/s [43], it uses a time-triggered communication method with characteristics as deterministic and fault-tolerant with its two independent data channels. Except for x-by-wire systems, it is also interesting for the safety-critical and real-time system-related field in advanced automotive control applications.

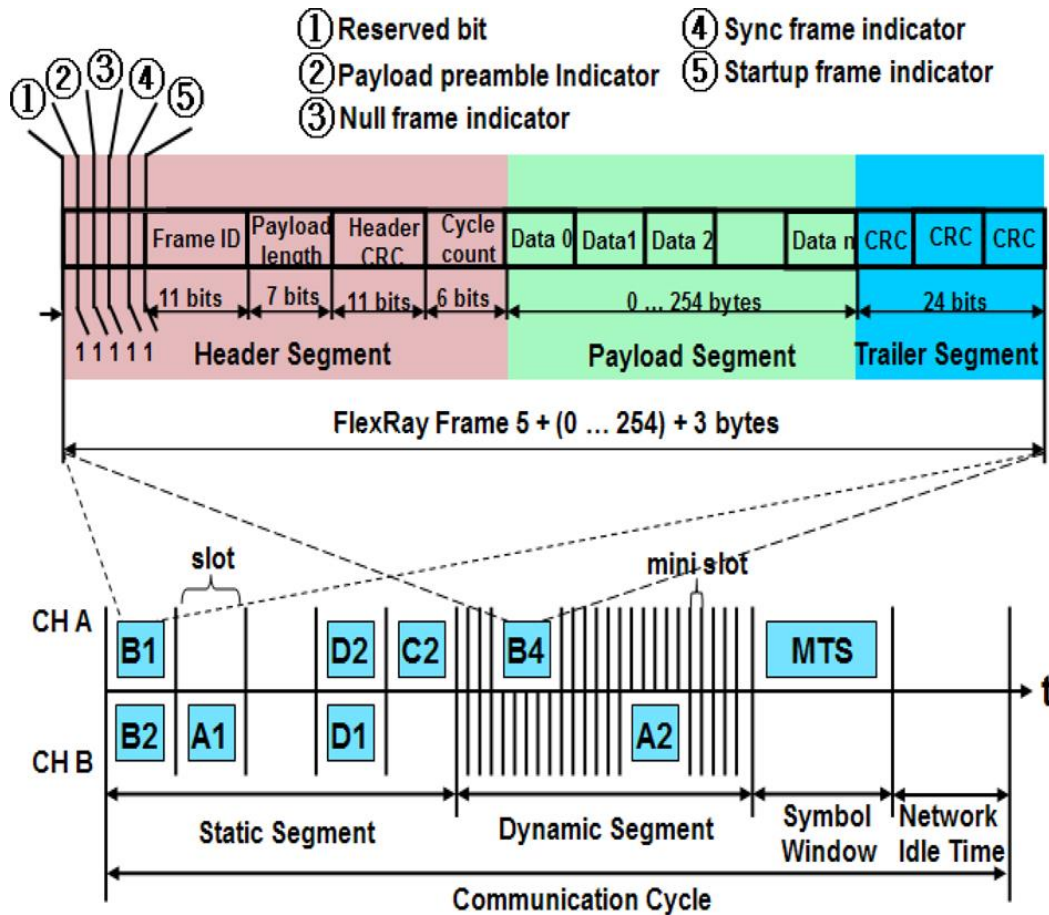


Figure 2-4: FlexRay data frame [41]

FlexRay ensures the transport of extraordinarily large quantities of data within the active chassis system in an extremely short period and reliably between the central control module and the ECUs [44]. And for that it uses depicted cycles. Where one communication cycle consists of four segments: static segment (ST), dynamic segment (DYN), symbol window (SW) and network idle time (NIT) as shown in Figure 2-4 [41].

A FlexRay frame consists of three segments: header segment, payload segment and trailer segment as represented in Figure 2-4. The first five bits are defined as the basic features of the frame. Frame ID (11 bits) is defined as the slot position in the static segment. For the dynamic segment, frame ID is used to define the priority of the frame: a lower identifier indicates higher priority. Payload length (7 bits) is defined as the data length (two times the payload length minus the number of data bytes). Header CRC (11 bits) is a cyclic redundancy check, which is computed over the Sync frame indicator (1 bit) is the serial number of the frame defined locally in the node. Payload segment (0 to 256 bytes) contains main data. Trailer segment (24 bits) is for cyclic redundancy check, which is computed over the header segment and the payload segment [41].

2.3.4 CAN Protocol

To overcome the limitations and the weakness of centralized control system fieldbus communication systems were developed answering difficulties such as modification/extension, extensive wiring and high installation costs. The limitation of fieldbus systems lies mainly in transmission expansion, a limited variety of topologies and transmission media. These limitations can be overcome by a network-based control system that distributed real-time control is possible [45].

The CAN protocol was originally developed to satisfy distributed real-time control needs in automotive applications. The use of CAN technology has been extended to other custom applications, including industrial control applications. Various application layers have been developed with Specifications Specifically oriented to industrial and process control applications, control networks for heavy-duty trucks and buses, distributed control Systems, and control networks for cars [46].

The CAN technology is described by six ISO documents:

- **Part 1:** Data link layer and physical signaling;
- **Part 2:** High-speed medium access unit;
- **Part 3:** Low-speed, fault-tolerant, medium-dependent interface;
- **Part 4:** Time-triggered communication;
- **Part 5:** High-speed medium access unit with low power mode;
- **Part 6:** High-speed medium access unit with selective wake-up functionality.

However, for the beginning there is only the need to focus on the first three parts. Where the Part 1 describes the CAN protocol, it specifies the Classical CAN frame format and the newly introduced CAN Flexible Data Rate Frame format. The Classical CAN frame format allows bit rates up to 1 Mbit/s and payloads up to 8 byte per frame. The Flexible Data Rate frame format allows bit rates higher than 1 Mbit/s and payloads longer than 8 byte per frame [47]. It also covers the logical link control (LLC) sub-layer, medium access control (MAC) sub-layer and physical coding (PLS) sub-layer. Describing up to three implementation methods fusing the use or not of the standard CAN with CAN flexible.

Part 2 specifies the high-speed physical media attachment (HS-PMA) of the controller area network (CAN), a serial communication protocol that supports distributed real-time control and multiplexing for use within road vehicles. This includes HS-PMAs without and with low-power

mode capability as well as with selective wake-up functionality [48]. Finally, Part 3 [49] specifies characteristics of setting up an interchange of digital information above 40 kBit/s up to 125 kBit/s. This part of ISO 11898 describes the fault tolerant behavior of low-speed CAN applications, and parts of the physical layer according to the ISO/OSI layer model.

A CAN network consists of several CAN nodes which are linked via a physical transmission medium (CAN bus). In practice, the CAN network is usually based on a line topology with a linear bus to which several electronic control units are connected via a CAN interface. The passive star topology may be used as an alternative [50].

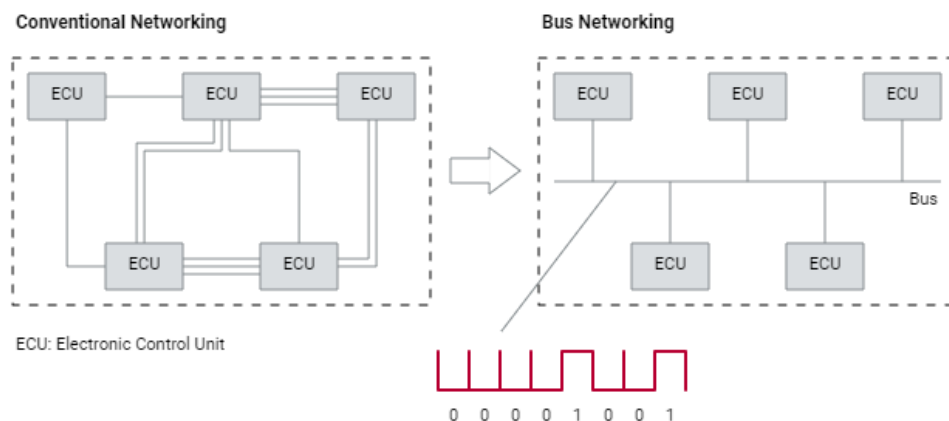


Figure 2-5: Conventional Networking vs CAN Bus Networking [50]

A CAN bus communication has solid foundations that lay on some very important principals [50]:

- **Decentralization** - Safety-critical applications, such as those in the powertrain area, place severe demands on a communication system's availability. So, it would be disadvantageous to assign responsibility for bus distribution to just a single bus node. Failure of this vulnerable bus node would cause all communication to fail. A much more elegant solution is to decentralize bus access, so that each bus node has the right to access the bus.
- **Event-Driven** - That is why a CAN network is based on a combination of multi-master architecture and line topology: essentially each CAN node is authorized to place CAN messages on the bus in a CAN network. The transmission of CAN messages does not follow any predetermined time sequence, rather it is event-driven. The communication channel is only busy if new information needs to be transmitted, and this allows for very quick bus accesses.

- **Receiver-selective addressing** - A method of receiver-selective addressing is used in a CAN network to prevent dependencies between bus nodes and thereby increase configuration flexibility: Every CAN message is available for every CAN node to receive (broadcasting). A prerequisite is that it must be possible to recognize each CAN message by a message identifier (ID) and node-specific filtering. Although this increases overhead, it allows integration of additional CAN nodes without requiring modification of the CAN network.

To ensure a high level of availability and reliability of the information being transferred in the CAN bus there are also several techniques applied to its protocol. And so, to detect corrupted messages, the CAN protocol defines five mechanisms: **Bit Monitoring** (every bit sent is automatically read from the bus line to ensure compatibility), **Form Check** (monitoring of the message format), **Stuff Check** (monitoring of the bit coding), **ACK Check** (evaluation of the acknowledgement) and **Cyclic Redundancy Check** (verifying the checksum). The bit monitoring and ACK check error detection mechanisms are performed by the sender. Independent of acceptance filtering, the receivers perform the form check, stuff check and cyclic redundancy check.

The use of a single line for every node means that every node needs to be able to respond within a specific time frame defined in CAN 2.0. Thus, there is a maximum bus line length that each baud rate can have, this relation is represented in the Figure 2-6.

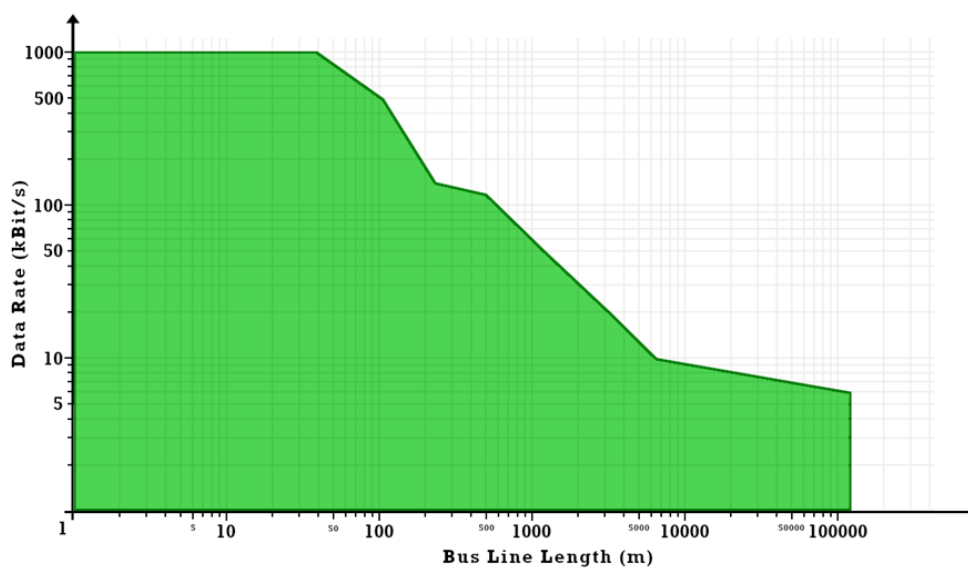


Figure 2-6: Data rate relation with the bus line length

2.3.4.1 CAN Classic

CAN classic is the first iteration of the CAN protocol before the development of CAN-FD. With its ISO's described in the previous section it can support baud rates up to 1Mbps and really slow communications in order to increase range of the network. It also supports different types of message frames depending on the objective of the message.

The CAN data frame is the most important frame since it is the one responsible the transmission of data between different nodes. It is in this frame that different aspects of the frame are defined, such as the message length and the type of ID. Each segment has its proper name and different functions, as shown in Figure 2-7:

- **Start of frame (SOF);**
- **Identifier (ID)** can either have 11 or 29 bits long depending on being a standard or extended ID message where lower values have higher priority;
- **Remote Transmission Request (RTR)** indicates whether a node sends data or requests dedicated data from another node;
- **Control Field (CTRL)** contains the Identifier Extension Bit (IDE) and the Data Length Code (DLC) that specifies the length of the data bytes to be transmitted (0 to 8 bytes) and the reserved bit for future improvements to the protocol;
- **CRC Field** containing a fifteen bit cyclic redundancy check code used to ensure data integrity;
- **Acknowledge Field (ACK)** indicates that at least one node in the network which has acknowledged and received the data correctly;
- **End of Frame (EOF);**
- **Inter Transmission Message (ITM);**

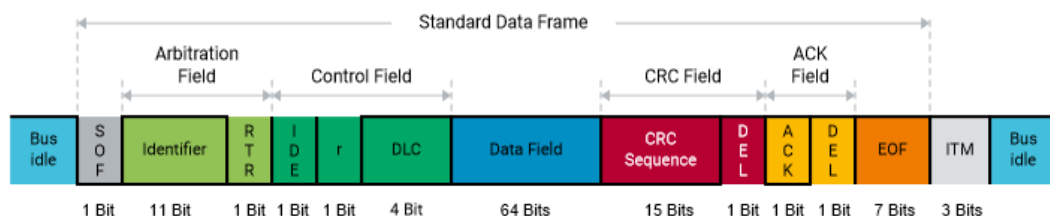


Figure 2-7: CAN Classic data frame

A CAN remote frame are used in polled networks with the objective of requesting a particular message to be put on the message. However there it requires that a specific node on that network to be ready to receive this type of message [51]. The message layout is the one in Figure 2-8.

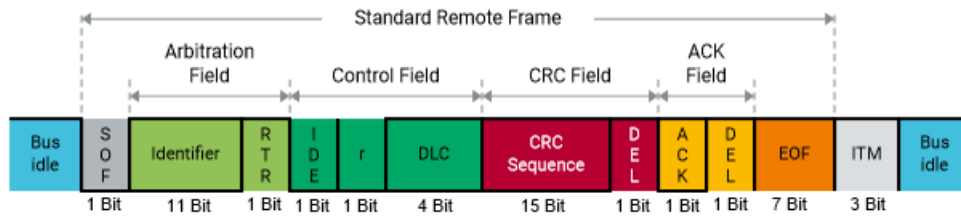


Figure 2-8: CAN remote frame

The CAN error frame is a special frame that is sent to the bus when a frame has been detected to have an error (one or multiple of the described above in section 2.3.420). This type of frame will cause an error to every node to guarantee that the previous transmitted frame encountered an error and the network is notified.

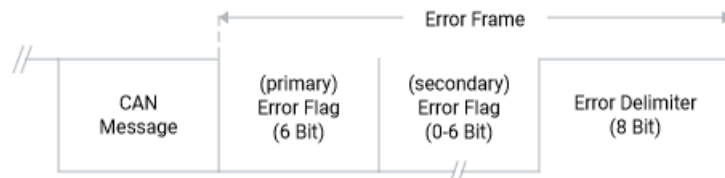


Figure 2-9: CAN Error frame

2.3.4.2 CAN-FD

CAN with flexible data rates was the response from the CAN protocol to answer automotive harsher bandwidth requirements and explosion in data in vehicles. Keeping the strengths of CAN but improving the throughput up to six times and diminishing the protocol overhead. To achieve this there were made some small changes to the layout of the data frame (shown in Figure 2-10), while keeping the error frame the same and removing the support for the remote frame, the notable changes to its layout are the addition of different bits and the function change of others:

- **Remote Request Substitution (RRS)** since remote frames are not supported at all this bit is always dominant ("0").
- **Flexible Data Frame (FDF)** is the bit indicating the use of CAN FD data frame.
- **Bit Rate Switch (BRS)** can be dominant ("0"), meaning that the CAN FD message data is sent at the arbitration rate or that is sent at a higher bit rate.
- **Error Status Indicator (ESI)** bit is by default dominant ("0") or error active. If the transmitter becomes error passive is indicated by being recessive ("1").

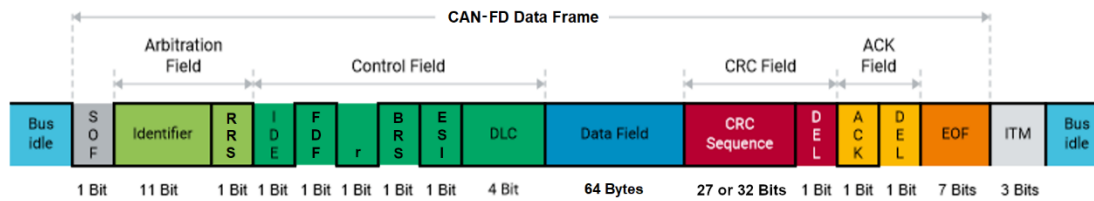


Figure 2-10: CAN-FD data frame

Other improvements can be described in terms of [52]:

- **Increased Length** – support up to 64 bytes per data frame reducing the protocol overhead and improving the data efficiency from around 50% to up to 90%.
- **Increased Speed** – support for dual bit rates one for the arbitration (nominal) up to 1Mbps and the data bit rate that supported up to 5Mbps by the ISO, however can achieve higher speeds depending on the network topology and transceivers.
- **Better reliability** – improvements of the cyclic redundancy check which lower the risk of undetected errors.
- **Smooth transition** – CAN and CAN FD ECUs under some specific conditions which allows for a certain integration of CAN FD nodes making it less expensive to transition for OEMs.

2.3.4.3 CAN Applications

Recently, in many systems of various application areas, such as airplanes, cars, building automation, and industrial automation systems, the network-based control system using fieldbus has been introduced. The network-based control system is usually composed of controllers, sensors, and actuators. The network-based control system can execute efficiently mutual functions between network components, such as multiple real-time controls and the exchange of information. Also, sensor signals and control signals generated by the network components are required to be transmitted in real-time to the corresponding network nodes [15] [23].

And it was among this necessity that the development of systems X-by-Wire started more than two decades ago, first within the military then it was adapted to commercial airplanes and only more recently into terrestrial vehicles [51]. The throttle-by-Wire system is a system that has been widely accepted by the automotive industry in contrast to the Brake-by-Wire and Steer-by-Wire systems that only brands like Mercedes and Toyota have decided to integrate these technologies into a small group of models of their cars. All these systems use backup strategies so that in the event of a main system failure, a secondary system can be entered to ensure the task is performed.

To fully grasp what is a Steer-by-Wire system, one must go back to more conventional systems. The manual steering system, shown in Figure 2-11, is the oldest form of steering where the steering wheel controls are mechanically used to drive the front wheels [52]. However, it is particularly hard to maneuver. Improvements come in the form of what is now called the power steering, shown in Figure 2-11. While still giving enough feeling of the road it made maneuvering much easier. These make use of a hydraulic unit to assist the driver in turning moments of the steering wheel. When the difference between the measured steering wheel torque (driver direction) and wheel torque is greater than a certain value, hydraulic fluid is released into the system, making steering more comfortable. Currently, hydraulic systems are mostly used by trucks or heavy vehicles [53], but most land vehicles use electric steering [52]. This is very similar to the hydraulic system but uses an electric motor to adjust the torque.

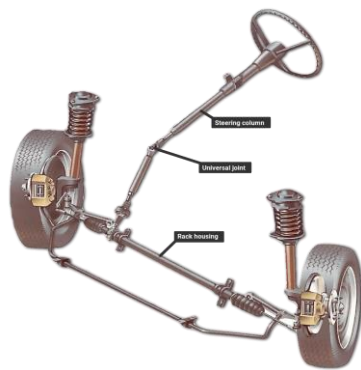


Figure 2-5a: Manual steering [54]



Figure 2-5b: Hydraulic power steering [55]



Figure 2-5c: Electric power steering [55]

Figure 2-11: Steering wheels different implementations. With early implementations of a purely manual steering wheel (5a) and future implementations using hydraulic parts (5b) and electronic motors(5c)

The attempt to replace the traditional steering system with the SbW system occurs in the 1990s [52]. A traditional SbW system can be subdivided into three subsystems, shown in Figure 2-12: hand wheel, front tires and electronic control unit [51]. Where in the hand wheel there is the responsibility of measuring the steer angle and providing feedback to the driver of the feeling of the road. The second subsystem consists of an angular sensor and a motor responsible for changing the direction of the front wheels. The last subsystem concerns the ECU, which has hardware and software components capable of analysing incoming data and making decisions regarding vehicle steering. This is just a possible application where the transmission and assurance of the data transfer is crucial for the correct operation of the whole system.

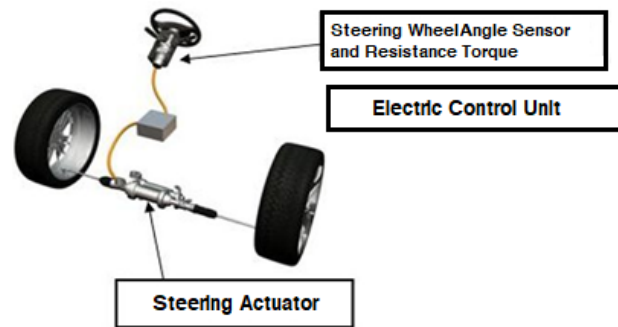


Figure 2-12: Steer-by-Wire Steering

Other important applications brought by the CAN FD are in **Electric Vehicles** since their powertrains require far higher bit rates with added complexity with new control units related to the DC/DC inverter, battery, charger, range extender and others. Ability to **ECU flashing** with higher bit rates that allow for software updates and upgrades via CAN. Applications in real time **robotics** such as multiple axis arm movements transitioning from CANOpen to CAN FD with its increased efficiency. Increasingly, **Advanced Driver Assistance Systems (ADAS)** are being introduced in passenger cars and commercial vehicles. This pressures the bus load of Classical CAN, yet ADAS is key to improving safety. Here, CAN FD will be key to enhancing safe driving in the near future. And the ability to prevent hacks on the CAN buses by **securing CAN Bus** by implementing CAN FD authentication via the Secure Onboard Communication (SecOC) module may be a key roll-out driver.

2.4 Conclusions

This chapter gave an overview of topics related to development practises in embedded systems for safety and reliability. It also dove deep into different protocols of communication with an emphasis on automotive and fault-tolerant protocols with different safe guards. It ended with an understanding of CAN protocol and its successor the CAN-FD protocol which are the main focus of this dissertation.

To summarize, this chapter builds the basic blocks to understand the future decisions made during the creation of the dissertation, as well as, the guidelines and standards used in it, to ensure a reliable and safe communication driver.

Chapter 3

System Specification

Chapter 2 describes technology available in safe embedded communication and its principles. It also describes methods and requirements for its development and testing stages.

To add context, the main goal of this dissertation is to develop a safe driver of CAN communication following standard guidelines, which can be used in any type of application with the mentality of plug-and-play.

This chapter will describe all the design, architecture, and specification of the system. Starting by describing all requirements and constraints required by the stakeholder and further down the implications they have on the requirements of the system. Followed by the detailed AUTOSAR based system architecture and its brief description and explanation. Finally, the last part of this chapter describes all the hardware used and its possible variations supported.

Summarizing, taking in consideration the principles studied in Chapter 0 it is given a general overview of the whole system architecture.

3.1 Project Requirements

To properly design and conceive the system, it is crucial to define all requirements and constraints in advance, as it is essential to fulfil them for the correct operation of the system. The project requirements are divided into two groups: the stakeholder group that defines the basis of the project and facts as the basis for the second group of system requirements, with more specific and clear requirements.

Stakeholder Requirements

ID	Description	Priority	Owner
SKH-1.	CAN driver shall be created to fit in an AUTOSAR based system architecture with Steering Angle Sensor (SAS)	High	Bosch
SKH-2.	The CAN driver shall be implemented within the family of the NXP devices S32Kxxx	High	Bosch
SKH-3.	The GUI shall be created using CANoe Vector tool and called LWS-Demo Panel	Moderate	Bosch
SKH-4.	The GUI shall be composed of several panels, being able to do demonstrations, configurations, simulations and calibrations for the Steering Angle Sensor (SAS)	Moderate	Bosch
SKH-5.	The GUI shall be able to receive, send and analyse CAN frames	Moderate	Bosch

Table 3-1: Stakeholder Requirements

System Requirements

ID	Description	Type	Derived From
SYS-1.	The system shall follow the AUTOSAR Stack name designation	Architecture	SKH-1
SYS-2.	The system shall be composed of a microcontroller (CAN capable), a sensor to provide data and a physical bus line	Architecture	SKH-1

SYS-3.	The transceiver shall differ depending on the microcontroller used	Architecture	SKH-2
SYS-4.	The system shall support the use of CAN transceivers or no CAN transceiver operations	Hardware	SKH-2
SYS-5.	The system shall be most independently possible from the transceiver used	Hardware	SKH-1
SYS-6.	The software layers above MCAL shall be work independently of it and therefore be universal for every devices	Software	SKH-1
SYS-7.	The software shall support sleep and stop operations	Software	SKH-1
SYS-8.	The software shall support wake-up from frame, pattern and interrupt	Software	SKH-1
SYS-9.	The software shall support polling and interrupt operations	Software	SKH-1
SYS-10.	The software layer of the transceiver shall isolate the operations for the transceiver configuration and operation.	Software	SKH-1
SYS-11.	The software shall be able to handle bus off event with a measured time frame	Software	SKH-1
SYS-12.	The software shall support Classic CAN and CAN FD	Software	SKH-1
SYS-13.	The software shall support messages from 0 to 64 bytes	Software	SKH-1
SYS-14.	The software shall support any message matrix and format within the size range	Software	SKH-1

SYS-15.	The GUI shall have the demonstration panel as the main panel	GUI	SKH-3
SYS-16.	The GUI shall support Classic CAN and CAN FD frames	GUI	SKH-5
SYS-17.	The GUI shall be able to simulate a working communication	GUI	SKH-4
SYS-18.	The GUI shall be able to reconfigure the CAN bus while running	GUI	SKH-4
SYS-19.	The GUI shall have messages for calibration, message type, manage CAN Power Boxes and software version.	GUI	SKH-4

Table 3-2: System Requirements

The stakeholders are responsible to form the basis of the system requirements, as well as the system validation. The whole GUI, named LWS-Demo Panel, has the intent of demonstration, simulation and tests, with the help of Vector tool CANoe as stated in the requirements SKH-3, SKH-4 and SKH-5 . It is possible to demonstrate the current steer angle or the different gear positions and their respective angles. Other important tasks are pre-configured messages to change the operation mode, sensor calibration commands, real-time CAN configurations, message filters and others.

The choice of the microcontrollers done previously comes as a requirement of the project, which means that all the features to be implement are directly or indirectly affected by this former decision (SKH-2). But since it is the same family means that the CAN controller in each device will be similar, resulting in only some adjustments for the MCAL which is the most dependent layer of the physical hardware implemented (SKH-2).

The decisions done by the stakeholders affect to some degree all the system requirements. The stack of the AUTOSAR architecture inspired the System Architecture stack with a normal CAN bus line (SKH-1). This physical layer will be dependent on if there are transceivers being used or if an approach without transceivers was chosen, both versions supported.

Onto the software, there are behaviours that are isolated for each layer, for example, the transceiver software component is responsible for the configuration and preservation of operation of the transceiver used (SYS-5). Since there are CAN bus transceivers with very different behaviours and supported features, ranging from simple conversions to CAN high and low lines to error detection, selective wake up, “ground shift detection” and many other features. It also must support different type of message with different payloads in Classic CAN and CAN FD (SYS-12). The handling of certain errors is a crucial and mandatory feature in which the bus off stands out. This operation is of crucial control, from its detection to the recovery of the CAN controller where the time it takes to go from the loss of operation to reinstating communication must be a very well-timed event, depending on the different application (SYS-11).

Following the stack, the bottom layer is where data and errors are stored and caught. Afterwards, they are propagated to the upper layers, where the decision for what to do with the errors and data is done and afterwards broadcasted downwards until the bottom layer. **It falls on the software to fill any shortcomings of the hardware**, but also to keep it in control be either functionalities or energy power efficiency. For this the driver supports sleep and stop operations as well different wake-up methods (either be pattern, frame or interrupt) even if the CAN controller does not support wake-up there are solution of transceivers that fit that roll (SYS-7 and SYS-8).

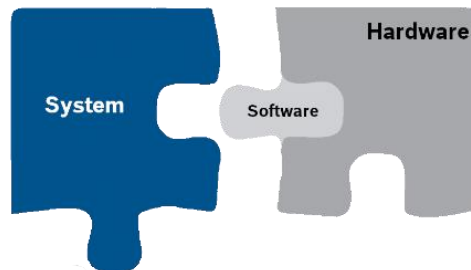


Figure 3-1: Representation of the relations between the system, software and hardware

3.2 System Architecture

As mentioned in the preceding sections, the goal is to develop the code (firmware) for a CAN communication protocol capable of handling end-to-end communication, from receiving, sending and other different features, such as wake up on CAN. Taking into consideration the previous study and all the requirements, it is possible to define and visualize the system components. Firstly, the system as most embedded systems are divided into two major layers: the hardware

layer and the software layer. The full system stack, containing all the different layer divisions and elements, is depicted in the Figure 3-2. The different elements that compose the hardware layer of the system are the microcontroller and the transceiver or capable layer for communication that will be addressed after this.

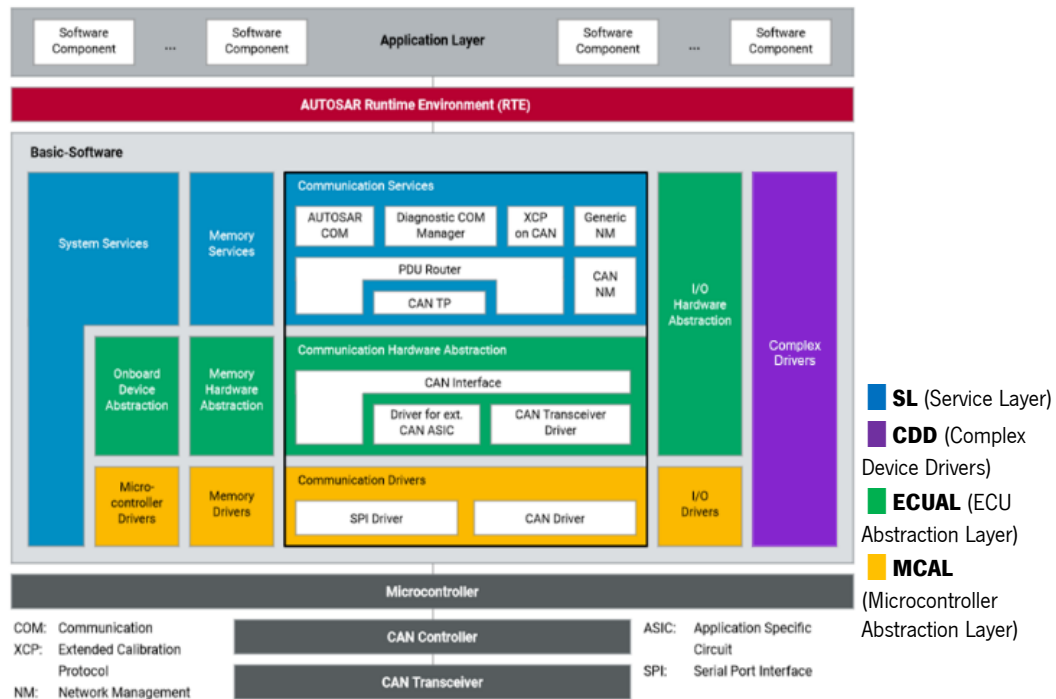


Figure 3-2: AUTOSAR Partial System Stack

Above this, there is the software layer. It is the microcontroller that will interface the hardware and the software layers, and thus will contain all the software elements. The software layer will follow closely the stack from AUTOSAR, the reasoning for this is because it is a software stack that was already built for having different layers for communications and obviously many other drivers. Above the “AUTOSAR layer” there will be the application layer. This layer normally holds the “users” configurations as well as other user applications.

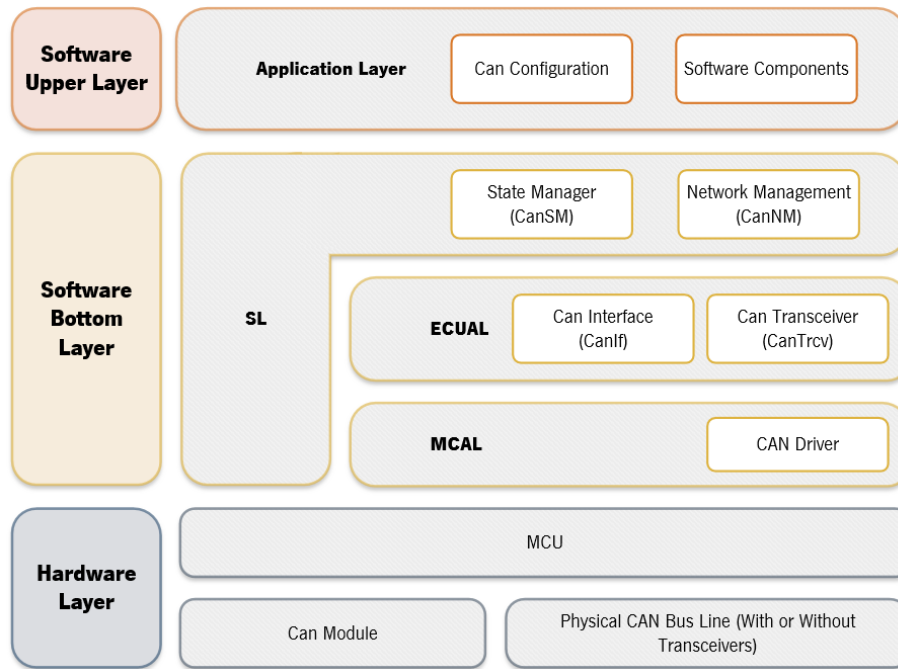


Figure 3-3: System Stack

The actual stack used for the software application was divided into the hardware layer and the software layer. Starting from the top the software layer was divided into two sub-layers, the upper one, is where lies the configuration done by the user and other software components that interact with sublayers, such as function wrappers. Function wrappers are APIs that serve to use basic and important functionalities of the CAN driver with ease, a good example would be the `printf()` function that is an abstraction layer to serial communication, where with only a function it is possible to send a message via COM without the hassle of configuring everything till then. The bottom software layer starts with the service layer responsible for the management of the state of the CAN controller and software, as for the Network Management it is more responsible for the node management and how is the connection to it currently. Under this layer lies the ECUAL responsible to interface the upper layers with the MCAL and with detailed configurations about the transceiver. Finally, the MCAL where the messages are transmitted, received, and errors are handled.

The hardware layer is composed of the microcontroller where the software runs and more specifically the CAN Module (with the CAN controller) and the CAN transceiver or physical layer responsible for the proper propagation of the CAN frames.

3.3 Hardware Specification

After describing the project requirements and the system architecture and stack, the Hardware Specification envisions to provide more detailed information about the hardware components of the system. As seen in Figure 3-3, the hardware layer is composed of MCU, and below it lays the two main components for CAN communication. The CAN module responsible for sending the message in the correct format as well as validating messages and their form, close to it there is the CAN physical layer. The physical layer can be defined into two types by the use of transceivers to create a differential two-wire communication bus line or by the non-use of transceiver, creating a simpler one-wire communication. This section will have a top-down approach from the MCU to the hardware responsible for the communication itself.

3.3.1 Microcontroller

The microcontrollers used were from the family of NXP microcontroller S32Kxxx, more specifically the S32K116 and S32K2TV (Test Vehicle for the next generation).

The S32K116 (shown in Figure 3-4) is a 32-bit general-purpose automotive microcontroller based on the Arm Cortex-M0+ core, with several key features.

- Operating characteristics:
 - Voltage range: 2.7 V to 5.5 V
 - Ambient temperature range: -40 °C to 105 °C for High Speed RUN mode, -40 °C to 150 °C for RUN mode
- Arm™ M0+ core, 32-bit CPU
 - Supports up to 112 MHz frequency (HSRUN mode)with 1.25 Dhrystone MIPS per MHz
 - Arm Core based on the Armv7 Architecture and Thumb®-2 ISA
 - Integrated Digital Signal Processor (DSP)
 - Configurable Nested Vectored Interrupt Controller(NVIC)
 - Single Precision Floating Point Unit (FPU)
- Clock interfaces
 - 4 - 40 MHz fast external oscillator (SOSC) with up to 50 MHz DC external square input clock in external clock mode
 - 48 MHz Fast Internal RC oscillator (FIRC)

- 8 MHz Slow Internal RC oscillator (SIRC)
- 128 kHz Low Power Oscillator (LPO)
- 32 kHz Real Time Counter external clock (RTC_CLKIN)
- Power management
 - Low-power Arm Cortex-M0+ core with excellent energy efficiency
 - Power Management Controller (PMC) with multiple power modes: HSRUN, RUN, STOP, VLPR, and VLPS
 - Clock gating and low power operation supported on specific peripherals
- Memory and memory interfaces
 - Up to 2 MB program flash memory with ECC
 - 64 KB FlexNVM for data flash memory with ECC and EEPROM emulation
 - Up to 256 KB SRAM with ECC
 - Up to 4 KB of FlexRAM for use as SRAM or EEPROM emulation
 - Up to 4 KB Code cache to minimize performance impact of memory access latencies
 - QuadSPI with HyperBus™ support
- Debug Functionality
 - Serial Wire JTAG Debug Port (SWJ-DP) combines
 - Debug Watchpoint and Trace (DWT)
 - Instrumentation Trace Macrocell (ITM)
 - Test Port Interface Unit (TPIU)
 - Flash Patch and Breakpoint (FPB) Unit
- Communications interfaces
 - Up to three Low Power Universal Asynchronous Receiver/Transmitter (LPUART/LIN) modules with DMA support and low power availability
 - Up to three Low Power Serial Peripheral Interface (LPSPI) modules with DMA support and low power availability
 - Up to two Low Power Inter-Integrated Circuit (LPI2C) modules with DMA support and low power availability
 - Up to three FlexCAN modules (with optional CAN-FD support)
 - FlexIO module for emulation of communication protocols and peripherals (UART, I2C, SPI, I2S, LIN, PWM, etc).

- Up to one 10/100Mbps Ethernet with IEEE1588 support and two Synchronous Audio Interface (SAI) modules.
- Timing and control
 - Up to eight independent 16-bit FlexTimers (FTM) modules, offering up to 64 standard channels (IC/OC/PWM)
 - One 16-bit Low Power Timer (LPTMR) with flexible wake up control
 - Two Programmable Delay Blocks (PDB) with flexible trigger system
 - One 32-bit Low Power Interrupt Timer (LPIT) with 4 channels
 - 32-bit Real Time Counter (RTC)
- ASIL B capable.

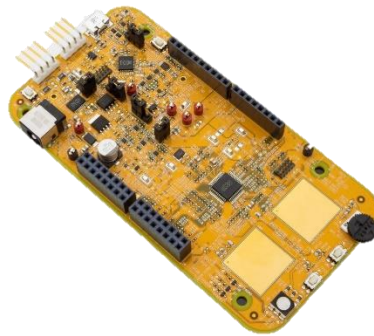


Figure 3-4: S32K116 development board

The S32K2TV (shown in Figure 3-5) is a much more powerful microcontroller with three 32-bit microprocessors one dual Arm core M33 and a M7 Arm cortex core.

- Operating characteristics
 - Voltage range: 2.97 V to 5.5 V
 - Ambient temperature range: -40 °C to 125 °C for all power modes
- Arm™ Cortex-M33/M7 core, 32-bit CPU
 - M7 supports up to 320 MHz frequency with 2.14DMIPS / MHz
 - M33 supports up to 160 MHz frequency with 1.5DMIPS / MHz
 - Arm Core based on the Armv7 and Armv8 Architecture and ThumbR-2 ISA
 - Integrated Digital Signal Processor (DSP)
 - Configurable Nested Vectored Interrupt Controller (NVIC)
 - Single Precision Floating Point Unit (FPU)

- Clock interfaces
 - 8 - 40 MHz Fast External Oscillator (FXOSC)
 - 48 MHz Fast Internal RC oscillator (FIRC)
 - 32 kHz Low Power Oscillator (SIRC)
 - 32 kHz Slow External Oscillator (SXOSC)
 - Up to 320 MHz System Phased Lock Loop (SPLL)
- Power management
 - Low-power Arm Cortex-M33/M7 core with excellent energy efficiency, balanced with performance
 - Power Management Controller (PMC) with simplified mode management (RUN and STANDBY)
 - Supports peripheral specific clock gating. Only specific peripherals remain working in low power modes.
- Memory and memory interfaces
 - Up to 4 MB program flash memory with ECC
 - Up to 256 K of flexible program or data flash memory
 - Up to 768 KB SRAM with ECC
 - Data and instruction cache for each core to minimize performance impact of memory access latencies
 - QuadSPI support
- Debug functionality
 - Serial Wire JTAG debug Port (SWJ-DP), with 2 pin Serial Wire Debug (SWD) for external debugger
 - Debug Watchpoint and Trace (DWT), with four configurable comparators as hardware watchpoints
 - Serial Wire Output (SWO)-synchronous trace data support
 - Instrumentation Trace Macrocell (ITM) with software and hardware trace, plus time stamping
 - CoreSight AHB Trace Macrocell (HTM)
 - Flash Patch and Breakpoints (FPB) with ability to patch code and data from code space to system space

- Serial Wire Viewer (SWV): A trace capability providing displays of reads, writes, exceptions, PC Samples, and print
- Full data trace for up to 16 output wide
- Embedded Cross Trigger (ECT) is used for multicore run-control and trace cross triggering, using CoreSight Cross Trigger Interface (CTI)
- Communications interfaces
 - Up to 20 serial communication interface (LINFlexD) modules, with UART and DMA support
 - Up to ten Low Power Serial Peripheral Interface (LPSPI) modules with DMA support and low power availability
 - Up to two Low Power Inter-Integrated Circuit (LPI2C) modules with DMA support and low power availability
 - Up to eight FlexCAN modules (with optional CAN-FD support)
 - FlexIO module for flexible and high-performance serial interfaces
 - One Ethernet module
 - 2-ch FlexRay module
 - Up to three Serial Audio Interface (SAI) modules
 - One Secured Digital Host Controller (SDHC)
- Timing and control
 - Up to three enhanced modular I/O system (eMIOS), offering up to 96 timer channels (IC/OC/PWM)
 - Up to three System Timer Module (STM)
 - Up to two Logic control units (LCU)
 - Full cross triggering support for ADC / timer (BCTU)
 - One Trigger MUX Control (TRGMUX) module
 - Up to four Periodic Interrupt Timer (PIT) modules
 - 32-bit Real Time Counter (RTC) with autonomous periodic interrupt (API) function
- ASIL B or ASIL D

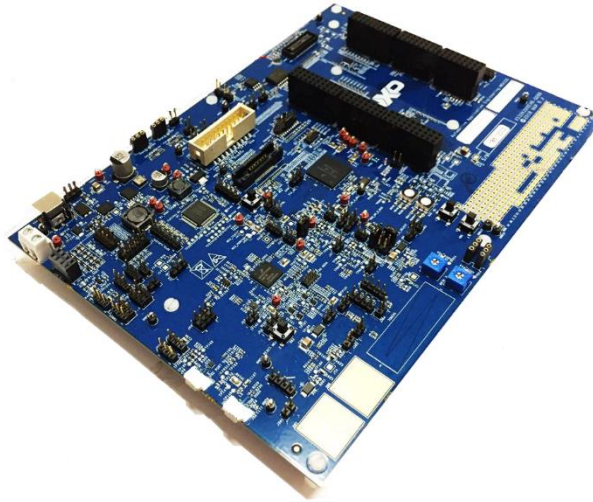


Figure 3-5: S32K2TV development board

3.3.2 CAN Module

A CAN Module (or CAN controller) is a hardware peripheral capable of sending and receiving CAN frames, while also handling all the error and form checking. On the NXP family, the module used is the FlexCAN module, which is a CAN protocol engine with a very flexible mailbox system for transmitting and receiving CAN frames. The mailbox system is composed of a set of message buffers (MB) that store configuration and control data, time stamp, message ID, and data. The memory corresponding to the first 38 MBs can be configured to support a Legacy FIFO reception scheme with a powerful ID filtering mechanism. This mechanism can check incoming frames against a table of IDs (up to 128 extended IDs or 256 standard IDs or 512 8-bit ID slices), with individual mask register for up to 32 ID filter table elements.

For Classical CAN frames, simultaneous reception through Legacy FIFO and mailbox is supported. For CAN FD frames, reception is supported through mailboxes and Enhanced Rx FIFO. For mailbox reception, a matching algorithm makes it possible to store received frames only into MBs that have the same ID programmed in the ID field. A masking scheme makes it possible to match the ID programmed on the MB with a range of IDs on received CAN frames. For transmission, an arbitration algorithm decides the prioritization of MBs to be transmitted based on the message ID (optionally augmented by 3 local priority bits) or the MB ordering.

The FlexCAN module is also able to receive and transmit messages in CAN FD format. The message buffers are sized to adequately store the quantity of data bytes selected by the FD control fields. The quantity of FD MBs available for a given quantity of data bytes is described in the FD control register. [56]

And for the most part the CAN modules between the families are basically equal. Discounting the CAN FIFO that are not going to be used the differences are listed in the Table 3-3.

Characteristic	S32K116	S32K2TV
Number CAN Channels	1	8
Number of message buffers	32	32
Maximum baudrate	8 Mbps	8 Mbps
CAN-FD	Yes	Yes
Partial Network	Support	No support
External Time Tick	Yes	Yes

Table 3-3: Differences between CAN modules

3.3.3 CAN Transceiver

The most basic capability of a transceiver is translating the Tx and Rx lines of CAN into two differential lines called CAN High and CAN Low, thus implementing a CAN bus line. They can achieve different maximum transfer speeds while also adding a certain delay that is transceiver dependent.

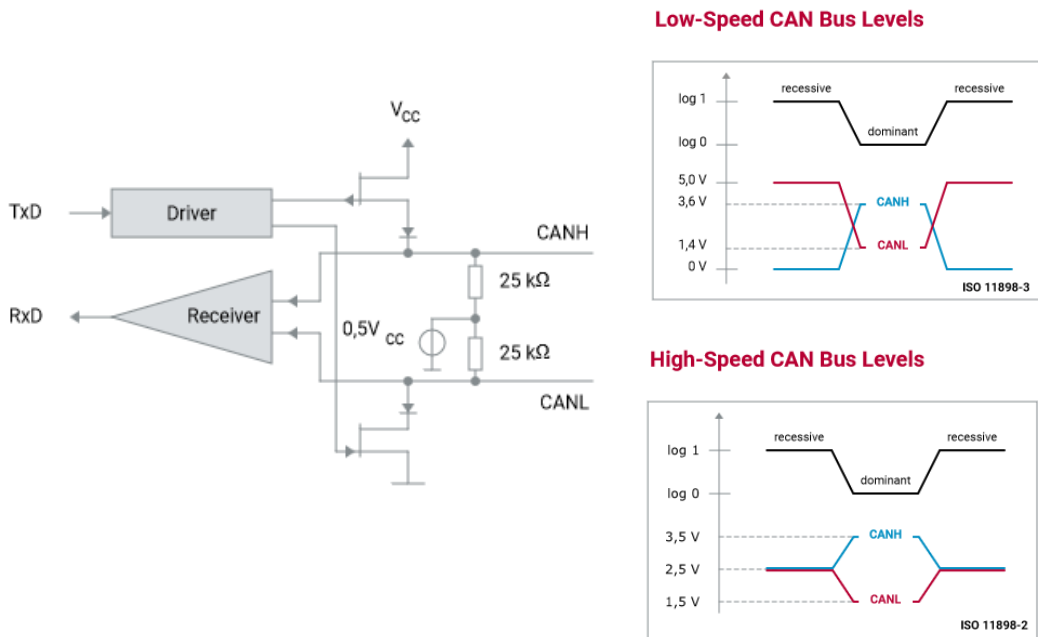


Figure 3-6: CAN transceiver

3.3.3.1 Selective Wake-Up CAN transceiver

This selective wake-up can be performed in two ways, by forming a wake-up pattern or a wake-up frame, being that the latter only works in classic CAN. This wake-up can also be triggered by a system interrupt via wire, and then scaled to the transceiver and/or CAN module.

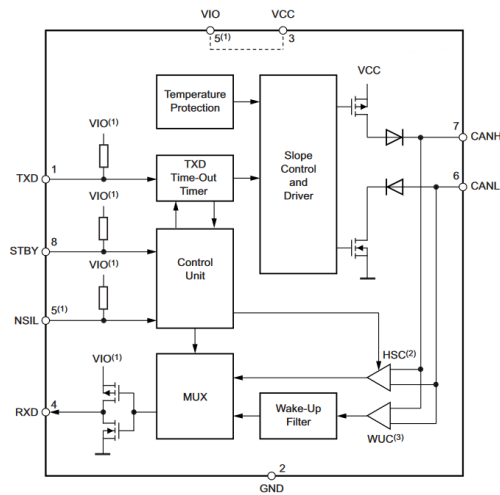


Figure 3-7: Transceiver with Wake-Up

3.3.3.2 Isolated CAN transceiver

An isolated CAN transceiver is expected to have an isolation barrier between the CAN Tx and Rx lines and the CAN High and CAN Low lines. This extra isolation layer is used for safety and preventing unwanted currents to flow in case of some type of flaw.

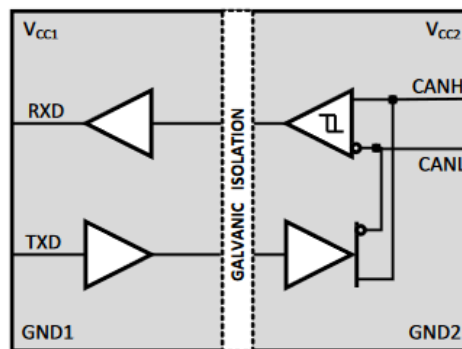


Figure 3-8: Transceiver with Galvanic Isolation

3.3.4 CAN Transceiverless

A no transceiver approach is based on connecting directly both Rx and Tx lines into a bus line without creating the differential lines of CAN High and CAN Low.

This type of architecture can work for distances below one meter. However, by decreasing the distance between CAN controllers it lowers the possible noise disturbance in the lines and increases the possible bit rates. To reduce the noise and disturbance it is possible to add an isolation barrier.

The diodes, represented in Figure 3-9, assure the correct direction of the current in the lines as well as the correct state of them, since the Tx lines must perform as open drains. Adding to this hardware, a voltage regulator may be required to use the pull-up resistor, with voltage typically being either 3V or 5V, depending on the TTL of the microcontroller and implemented baud rates [57].

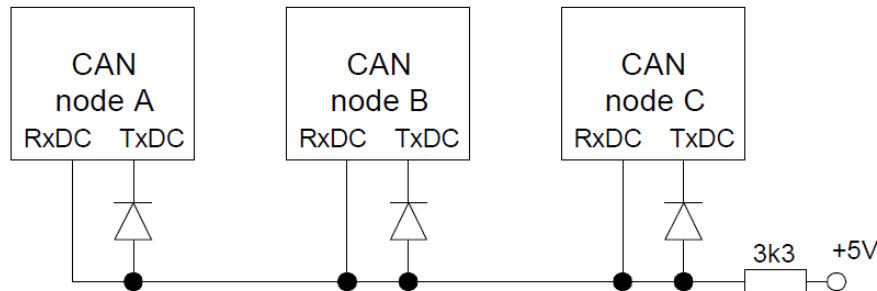


Figure 3-9: Connection of CAN nodes without CAN transceiver [58]

3.4 Conclusion

A good analyse aims to break down bigger goals to examine and better understand smaller parts and tasks. That is why it is of great importance to start by defining the requirements of the project. Starting by the stakeholders and going down to smaller parts of the system. On contrary of many systems, normally there is first the general list of the essential requirements and then there is the choice of hardware and software implementation. On this case, the stakeholders chose previously the microcontroller family and for that reason, some solutions were impacted. For example, the transceivers selection and the drivers wake-up functionalities. However, having the requirements listed makes the task of choosing which hardware and software to use simpler and swifter.

After selecting devices and hardware components the development phase can be initiated. Being possible to start designing and creating the software for the CAN driver as well, the GUI where its functionality will be tested and demonstrated.

Chapter 4

Implementation

After defining all the system specifications and components required, it was possible to proceed to the implementation of a communication driver, specific to CAN and capable of answering all the requirements as well as a complementary graphical interface for posterior demonstration and validation.

Therefore, this chapter provides an overview of how the system came to be. Since the hardware was previously provided or required only small adjustments it will not be mentioned in this chapter heavily. The two main focus points will be the two bigger pieces of software. The CAN driver, which composes the bigger part of this work, and the graphical panels, which become a great asset for posterior test and validation.

The hardware mostly consisted of changing the connections between boards, or them, and the computer. There was a small modification in the physical bus line, mentioned in Chapter 3.

The software was developed around the AUTOSAR stack concept and structured as such. Creating a distinct and thoroughly designed stack is an important step in order for the code to be clearer and understandable while minimizing errors. On another hand, the demonstration panels were developed with the support of a specialized tool, the CANoe from Vector.

4.1 Software Implementation

This chapter section is composed of all the steps taken when developing the software. It includes the development environment, the software layers and all the GUI development. It will explain briefly about different guidelines used when writing code and little cares about cleanness and clarity of the written code. It will also explain the different roles of each software layer for a better understanding of the development as a whole.

4.1.1 Development Environment

The CAN driver software for the S32K116 was developed in a complimentary Integrated Development Environment (IDE) for automotive and ultra-reliable Power Architecture, the S32 Design Studio for Arm, using C programming language. This IDE allows developing, managing, building and debugging embedded software. The S32DS IDE is a straightforward development tool based on open-source software, including Eclipse IDE, GNU Compiler Collection (GCC), and GNU Debugger (GDB), with no code-size limitations that enable editing, compiling, and debugging of designs [59].

However, the driver was created to be completely independent of the IDE and ready for porting into a different IDEs and compilers. However, for the S32K2TV, it was developed in the S32 Design Studio for S32 platforms, from the same company, NXP semiconductors.

All the code developed for the driver was following guidelines from the MISRA-C. MISRA C is a set of software development guidelines for the C programming language developed by the Motor Industry Software Reliability Association (MISRA). The guidelines aim to facilitate code safety, security, portability, and reliability in embedded systems. MISRA has evolved into a widely



Figure 4-1: S32 Design Studio (left) and MISRA (right) logos

accepted model for best practices by leading developers in sectors such as automotive [60]. Complementing the MISRA-C Guidelines is the document Specification of C Implementation Rules [35], created by AUTOSAR.

The development boards contain an on-board programmer and debugger, the OpenSDA for S32K116. As well as, JTAG for flashing and debugging purposes. For the S32K2TV, a probe from NXP was used for flashing and debug using the Cortex Debug Connector with 10 pins and support for interfaces such as Serial Wire and JTAG.

The rules for development for the panels were a little less restricted. Nonetheless, rules such as correct commentary and clear and objective of the code were maintained. The tool for development of the demonstration panels was the CANoe from Vector.

CANoe is the comprehensive software tool for the development, test and analysis of individual ECUs and entire ECU networks. It supports network designers, development and test engineers throughout the entire development process – from planning to system-level test.

Versatile variants and functions provide the appropriate project support. Therefore, its versatile functions and configuration options are used successfully by OEMs and suppliers worldwide [61].



Figure 4-2: CANoe logo

As stated, CANoe is an extensive tool where its support of CAN protocol and graphical interfaces makes it a great platform for the development of demonstration panels.

The two key tools that will be used are Vector CAPL Browser and Panel Designer. Vector CAPL Browser is a text editor based on C with some restrictions, that makes the bridge between the graphical panels and the messages received in the CAN bus lines as well the users of the GUI. The GUI developed in Panel Designer is aimed to be the most user friendly as possible while packing different tools into it.

4.1.2 Software Guidelines

There are several guidelines that can be chosen for the same purpose, the most important aspect is to remain consistent, to choose a specific way to develop and stick to it through the whole development cycle, this improves readability and testability of the code created. One way to do it is to create a document with all the guidelines to be followed during the development, this document can include rules such as:

- **Naming conventions** – these can include different ways to name variable and functions, by using camel notation or underscore notation.
- **Coding Guidelines** - these guidelines must be followed during implementation and are intended to make readable and testable code. They can include the use of language subsets, use of defensive implementation techniques (for example for each

“if” statement there must be an “else” statement), enforcement of strong typing and others.

- **Design Principles** – these principles include a wide variety of design rules to be followed during development, such as the level of restriction when using pointers, the return of functions, the use of recursions, type conversions and many others.
- **Project structure** – which should reflect the layered software architecture of the project.
- **Software Modules** - collection of software files (code and description) that define certain software functionality present on an ECU.
- **Software Interfaces** - interface which describes an interface (header) file containing functions and datatype definitions.

4.1.3 Software Layers

As seen in the previous chapter the Figure 3-3 depicts the software layer, which could be divided into an upper and the bottom layer, where the upper layer focuses on user interface configuration and add-ons and the bottom layer into core features and functions. The upper layer is composed of the application layer, which per turn has the CAN configurations and other software components. The bottom layer is divided into three smaller layers each one with a purpose that will be explained in the next sections. These smaller layers are the Service Layer, the ECU Abstraction Layer and Microcontroller Abstraction Layer. On the IDE it looks something like in the Figure 4-3.

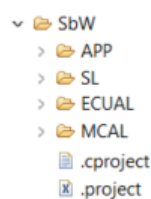


Figure 4-3: Project system stack folders

4.1.3.1 Application Layer

The application layer is the highest layer of abstraction in the software stack. It is home for any specific application components that will run on the top of the stack as well as ensuring proper communication between devices. On this case it is achieved by a series of wrappers that act as interfaces between the hardware and the communication protocol to send and receive CAN messages.

4.1.3.2 Service layer

The service layer for the CAN driver has two important modules the State Manager and the Network Management.

The first one implements the control flow for the CAN bus. It interacts with the Microcontroller Abstraction Layer and the Application Layer. On this layer, each network is defined with a unique network handle. Additionally, the CAN State Manager is responsible for the control flow abstraction of CAN networks.

The Network Manager has for its main purpose the coordination of the transition between normal operation and bus-sleep mode on a network. Beside, optional features such as detection of all present nodes or detection of nodes that are sleeping can be implemented.

4.1.3.3 ECU abstraction layer

The ECUAL similar to the service layer is composed of two components the CAN Interface and the CAN Transceiver.

The CAN Interface represents the interface to the services of the CAN Driver for the upper communication layers to manage different CAN hardware device types like CAN Controllers and CAN Transceivers used by the defined ECU hardware layout. It covers services as transmission requests, transmission confirmation, reception indication, controller mode, PDU control mode and others. Additional applications are the operation mode, via an interrupt, polling or even a mix of the two.

The CAN Transceivers is the module dedicated to the transceiver for which the controller is connected. There are several types of transceivers and each one can operate in a different manner, which increases the complexity of this module. To keep things simpler there were modules configured such as SPI ready for the eventual need for external configuration of the transceiver. There was also supported Standby mode in the transceivers, as well as watchdog features and partial network.

4.1.3.4 Microcontroller abstraction layer

The closest layer to the hardware is the MCAL and this layer is responsible for all the hardware access while making available for upper layers an extensive API capable of initiating transmission and calls the callback function of the CAN Interface module for notifying events, independently from the hardware. Furthermore, it provides services to control the behaviour and state of the CAN controller. It monitors Bus-off and Wake-up events and notifies them via callbacks. In summary, the

CAN driver is a software module responsible to access all the hardware resources directly connected to the CAN controller.

4.2 Software Workflow

The software workflow characterizes the different processes of operation of the CAN Firmware. From basic operations such as transmitting a message or receiving one, and the configuration of the working module to bus-off recovery and wake-up operations. This is a standalone module. This means that it will work without any other module attached. The process starts on the configuration file for the CAN, where is possible to configure each channel. Starting from the type of CAN that will operate (classic CAN or CAN-FD) to the velocity (data rates and respective sample point), the capture mode (polling or interrupt), transmission order, use of local priority and many more. The extensive configuration enables a more complex but flexible driver, where most configurations are compatible and if not an error is produced and the CAN channel not initialized.

4.2.1 State Machine

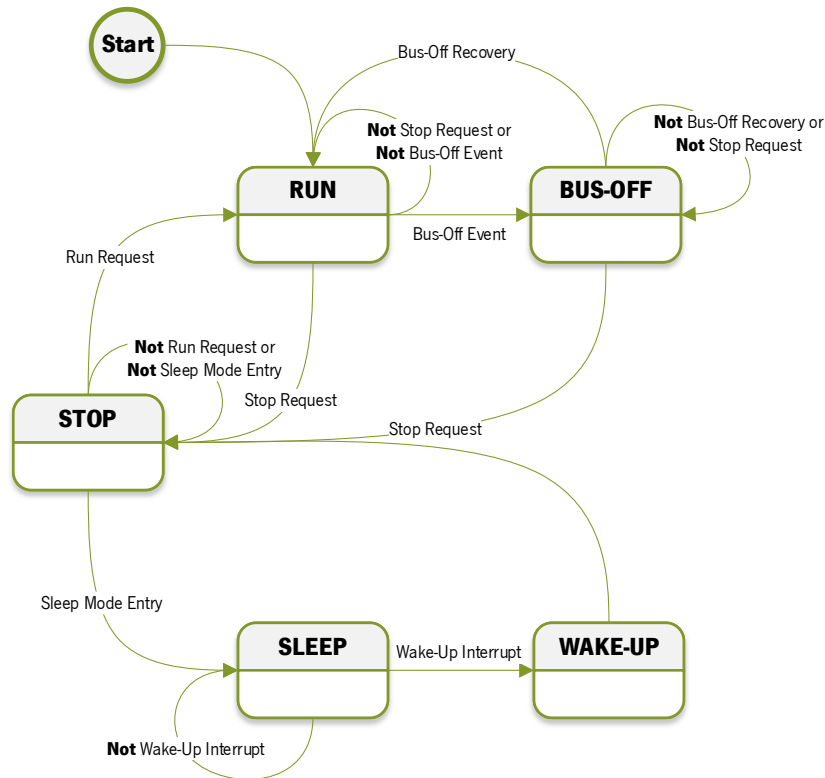


Figure 4-4: CAN Driver State Machine

A state machine allows linking the “state” of the system at a particular point in time and characterize the behaviour of the system based on that state, where a sequence of transitions driven by events and inputs describe the behaviour of the system [62]. In this case, the system is the CAN driver and its different operations, and the finite state machine in Figure 4-4 describes the behaviour of the system, allowing the state to know what operations the system is able to perform.

The operation of the CAN driver revolves around five states, and of those, it will stay for the most of its time in the RUN operation. When the module is configured and started the operation begins at the RUN mode, it is in this mode that the CAN driver is capable of sending and receiving messages. From there it can go to STOP state or Bus-Off state for the latter it would require a bus-off event to jump to it. This means that the communication has been cut and the module requires recovery to be able to send messages again, and is with exactly a bus-off recovery event that the state goes back to RUN.

When a stop request happens, the module stops sending messages and is a controlled event. It is from here that the module can transition to SLEEP state powering down the entire module

and normally most of the microcontroller. From the sleep more only a preconfigured interrupt can wake-up it and make a brief transition to WAKE-UP state before going back to STOP mode where can be started again.

4.2.2 Initialization

The initialization is a straightforward process. It starts with the load of the configurations for the correct channel, afterwards it checks for the type of CAN, between the slower classic CAN and the faster CAN-FD. There is also a pre-processing of the sample point before the advancing of the initialization. Having met all the conditions, the clock for the CAN is enabled and with it the enable for a stable configuration.

The next step would be to configure the data rate for that channel, which cannot change after configuration. Afterward, there is a memory cleaning and further configuration of the reception masks as well, general operation definition. The last two configuration before starting the module would be the configuration of the reception boxes and the partial network if the driver supports it and requires it.

Upon starting, there is a final check to see if the actual module started, and if not produce an error and finish the process.

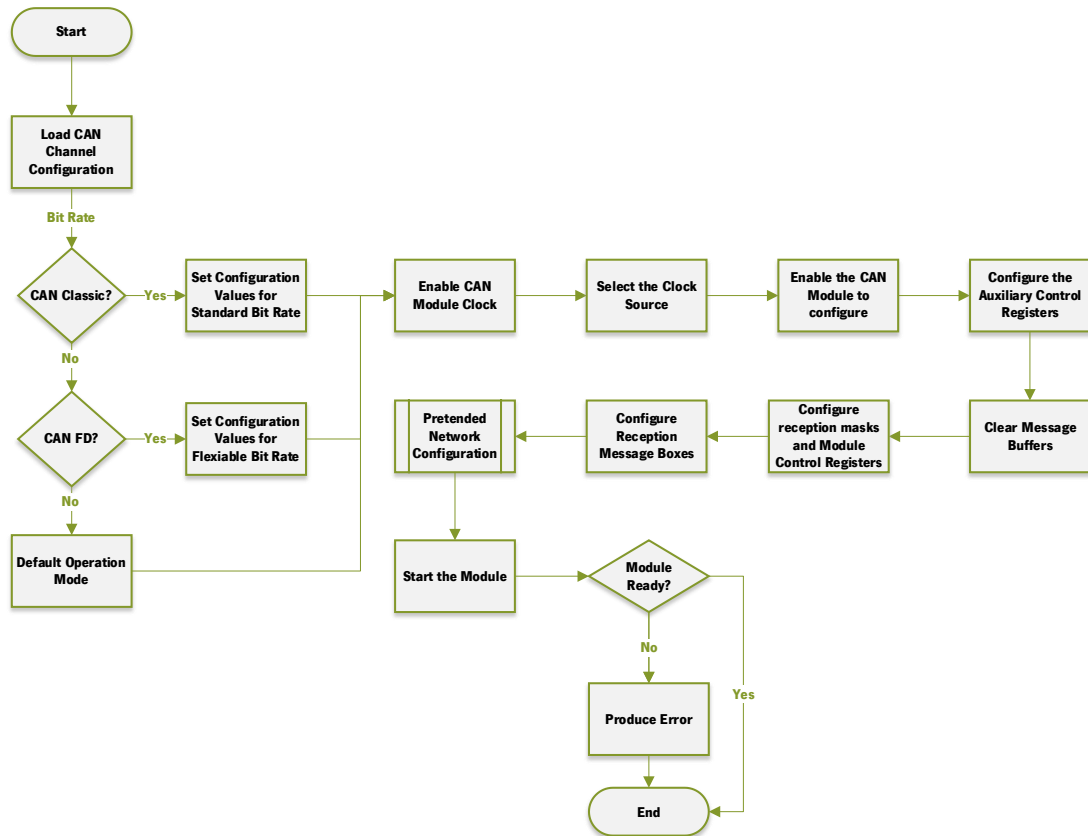


Figure 4-5: Initialization Flowchart

4.2.3 CAN wake-up

Summarizing wake-up is a feature where the CAN module has the ability to go from sleep (or standby) mode to normal operation through the capture of a frame or a pattern. It is depicted into three parts: Configuration, Sleep mode, and trigger Wake-Up through frame or pattern. The wake-up from the frame is a method only supported by CAN classic and it is also known as pretended networking (or partial network), since the module in this mode only has the ability to receive messages. The configuration of the pretended network as shown in Figure 4-6, is a process where several filters are set up for the ID, DLC, payloads as well the number of matches to trigger the wake-up with a safety time-out to make it leave sleep mode if no messages are being captured.

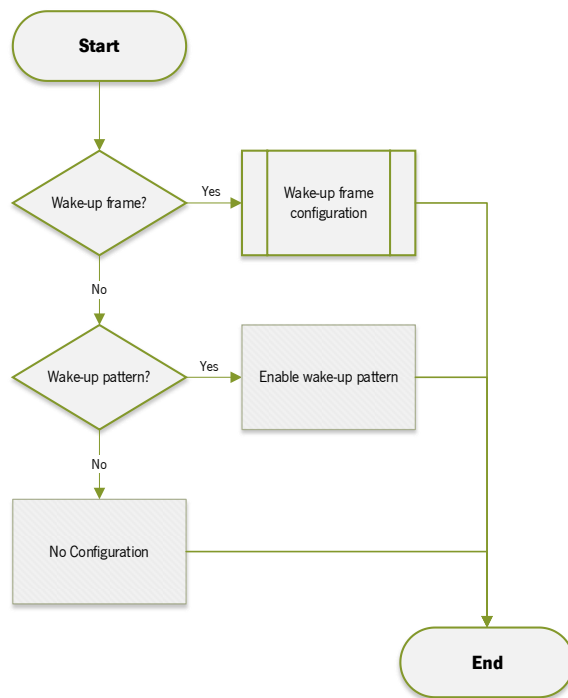


Figure 4-6a: Wake-up configuration flowchart

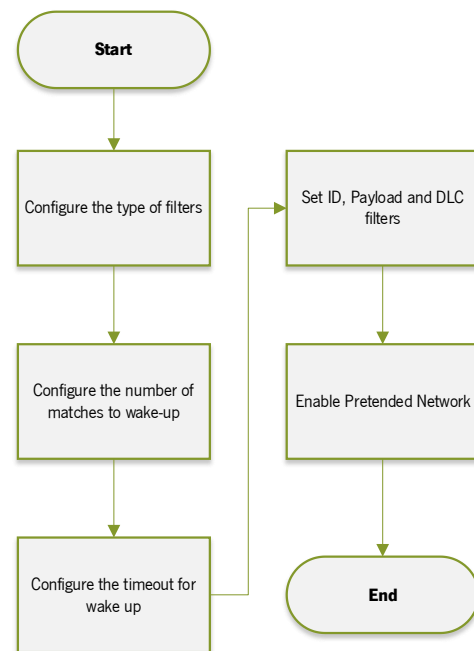


Figure 4-6b: Wake-up frame configuration flowchart

Figure 4-6: The wake-up process begins with configuration. There are two defined processes for wake-up via pattern (6a) which requires no extra configuration and wake-up via frame (6b)

To replicate this on CAN modules without neither of the wake-ups, this means that when sleep mode is enabled the entirety of the CAN module is turned off. There is a need to configure external transceivers with these capabilities. By norm, these configurations are done via SPI and much like on the controller there are a series of registers that allow a very similar configuration to the microcontroller. The transceiver detects the wake-up event, who then propagates it throw an interrupt pin to the microcontroller, waking it.

The second phase, after the configuration, is placing the microcontroller to sleep mode, this process is completely dependent on the microcontroller, but also on the microprocessor and how many of them. The common factor is disabling of the peripherals, in this case the CAN module is disabled beforehand and acts as the first step to sleep mode.

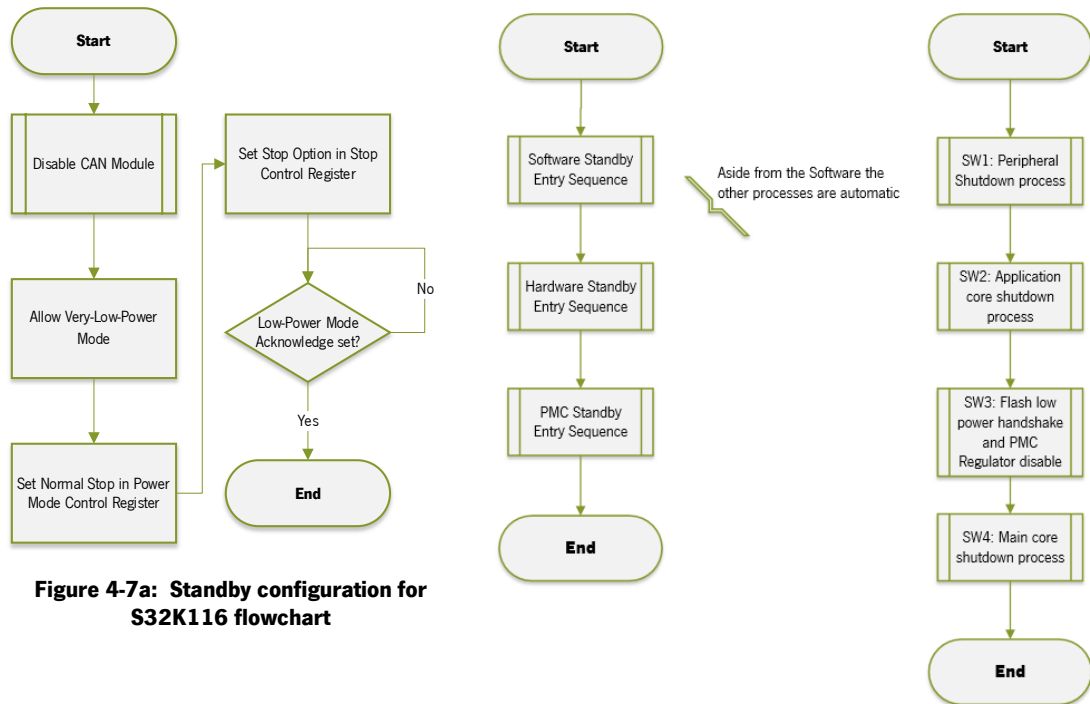


Figure 4-7a: Standby configuration for S32K116 flowchart

Figure 4-7b: Standby configuration for S32K2TV flowchart

Figure 4-7: The standby process differs within the family S32Kxxx because of the number of cores present in which one and the level of security enable in each one. Still the common ground starts by disable all the peripherals and start application shutdown.

In the S32K2TV, the process is more complex due to the presence of three microprocessors instead of one. This requires a higher level of coordination between cores and the present modules.

When the board is in sleep mode, the only thing that is left is to wait for a wake-up event. As stated before this can be in form of a frame or a pattern. Both are compliant with the ISO-11898-2:2016 [63] and therefore standard. The wake-up via frame requires the capture of valid frames that are posteriorly compared with the filters configured beforehand. If there is a match the count for wake-up is incremented once that count reaches the specified number a wake-up event is triggered and the microcontroller transitions to the previous running state.

The wake-up via pattern is broader in the fact that it allows for both classic CAN and CAN-FD frames to be the trigger for the wake-up event. The pattern has timed filters that go from dominant (logic “0”) to recessive (logic “1”) to dominant once again. The time it has to validate a dominant is filter dependent having two values for assessment, long filters from 0.5 to 5 microseconds and short filters with timings between 0.15 to 1.8 microseconds.

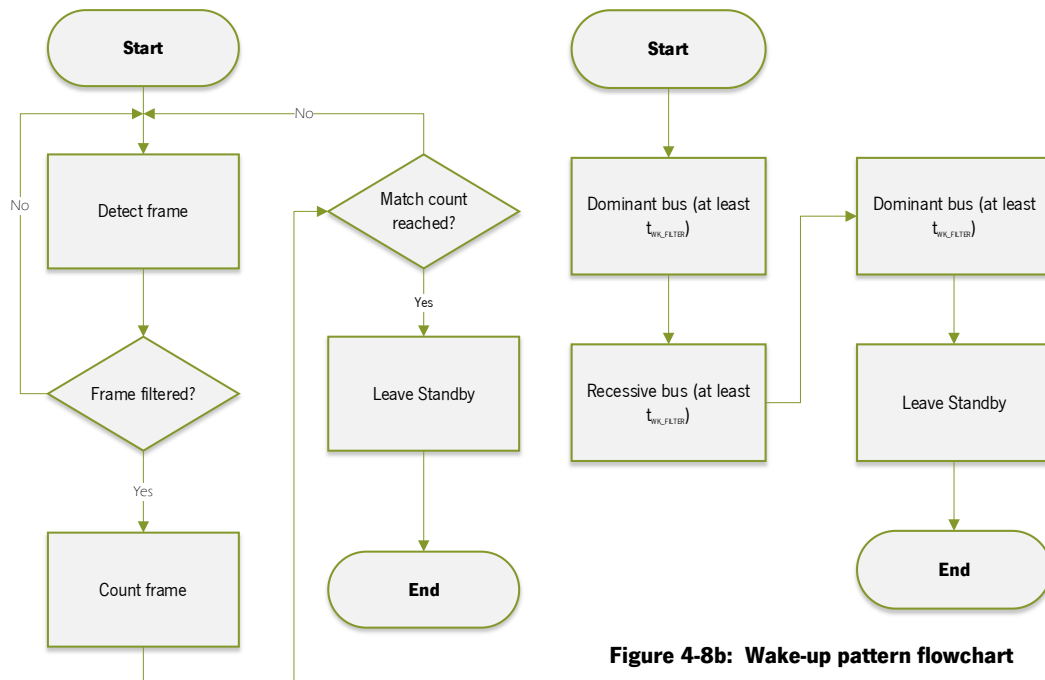


Figure 4-8a: Wake-up frame flowchart

Figure 4-8b: Wake-up pattern flowchart

Figure 4-8: Flowcharts describing the wake-up processes in order to leave standby, via wake-up frame and preconfigured messages (8a) and via wake-up pattern depicted in the ISO 11898:2-2016 (8b)

4.2.4 Bus-off Event

A bus-off event occurs when a transmission error overflows the 255 count. The recovery process has two possibilities automatically recover or with processor intervention. In the automatic process the node that is allowed to become error active, and therefore participate again in the bus arbitration process, after 128 occurrences of 11 consecutive recessive bits on the bus.

With manual recovery where the processor intervenes, the process is more controlled. However, it still must respect the timings of the recessive bits. There are several ways to handle this, logging the state, recover normally, clear messages boxes before recover and many other possible approaches. The chosen for this implementation passes through reconfiguration of the CAN module. This means a clean slate to further communications. In this process it is guaranteed that all the messages boxes are cleared, eliminating in the process the faulty message and possibly any error module associated. This configuration is described in the topic above in Figure 4-5. Only afterward, when the configuration is finished the actual bus-off recovery and reconnection to the line are done. By clearing the field called BOFFREC the recovery process is switched to automatic and with this, the count of recessive bits starts. Once it is completed, there

is a check for re-synchronization. Which marks the point, where it is again possible to participate in the bus arbitration process and send messages. After recovery, the BOFFREC is once again set (disabling automatic recovery) and the state of CAN is updated. To finalize the recovery, the flags signalling the bus-off occurrence and its recovery are cleared.

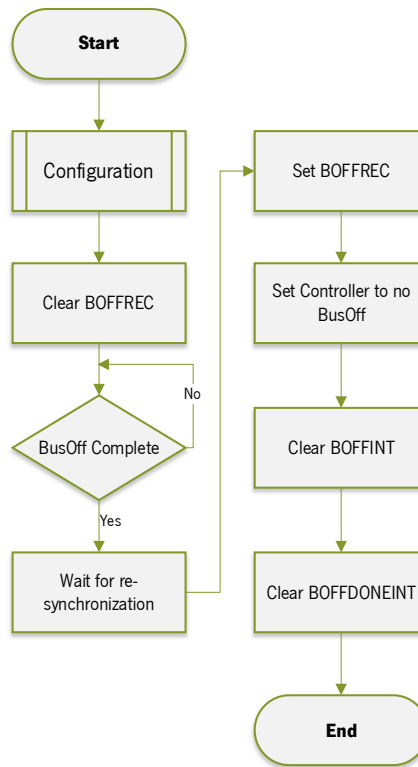


Figure 4-9: Recover from Bus-Off flowchart

4.2.5 Read and Write operations

As in most protocols, CAN protocol with its multi-master approach that allows for the transmission and reception of messages, each process has its own unique features, however, both occur in the same area of memory, a space dedicated to the division of message boxes capable of sending and receiving messages depending on how they are configured.

The writing operation described on the left flowchart of Figure 4-10, starts with the data to be transmitted and saving it. After it chooses a message box and verifies if it is already being used, if so abort the transmission and clear the message box. After having a clear message box, the data is registered and the header of the message of configured (DLC, ID, bit rate switch, and others). For the message to enter the arbitration process and compete for a spot on the bus it is required a specific code on the message box.

The opposite operation, and opposite for full communication capabilities, starts by running through all the receive flags to find if any message box flagged for the reception. When the flags are set, the process starts by checking the message and storing all the data, as well as the code, DLC, ID, timestamp and any other relevant information, as described in the Figure 4-10. All the data is saved into a PDU and the timer of the module is read in order for the message box to be available to receive a new message as soon as possible. After that, the flag is cleared and reception confirmation is sent across the layers.

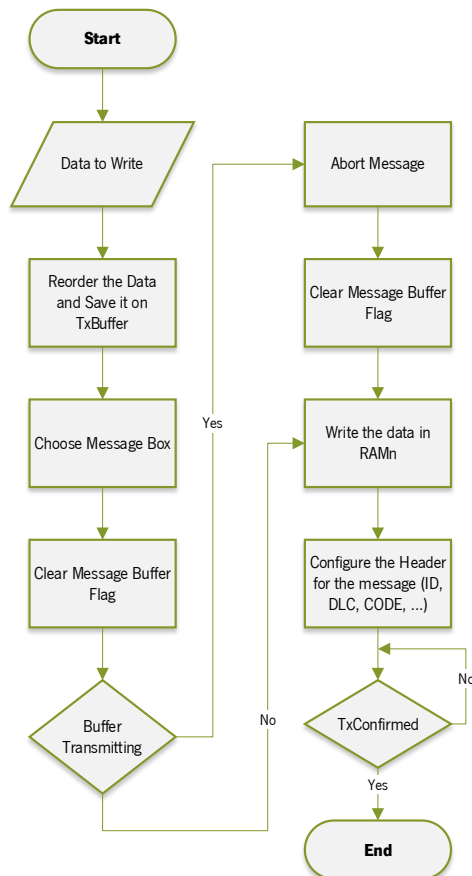


Figure 4-10a: CAN Transmission flowchart

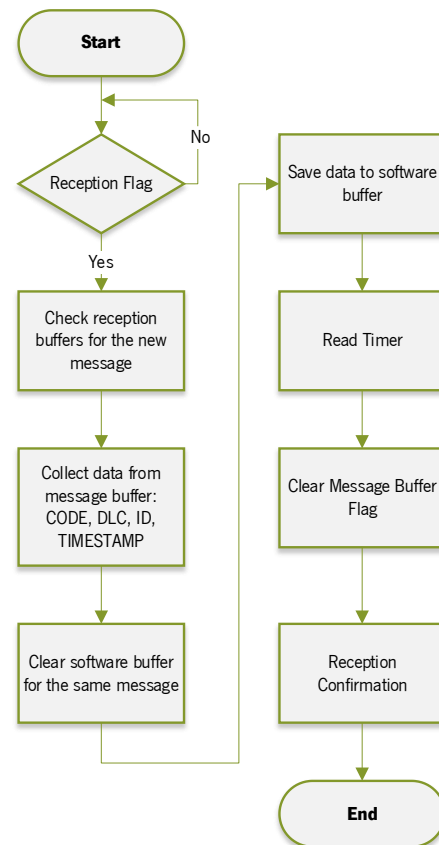


Figure 4-10b: CAN Reception flowchart

Figure 4-10: CAN main operations are the writing operation where there is the transmission of data to the CAN bus (Figure 4-10a) and the complementing operation of reading through the reception of frames (Figure 4-10b)

4.3 CANoe Software

All the software in CANoe is implemented using CAPL (Communication Access Programming Language) programming language. The GUI would use the help of a panel designer as the main block with support of API on CAPL for further operations.

4.3.1 CANoe Environment

Before starting coding and designing the CANoe Panels, it is important to understand the tools that will be used and the ones that will support the whole program. The base program allows for the creation of global variables to link events between the CAPL and the GUI, it also allows for graphical analysis in real-time, with extensive tooling for automation. Figure 4-11 shows the base environment required to start developing. Each ECU will have CAPL code and will have panels associated to them, combined with what CANoe calls environment variables that will bridge the Panels and CAPL.

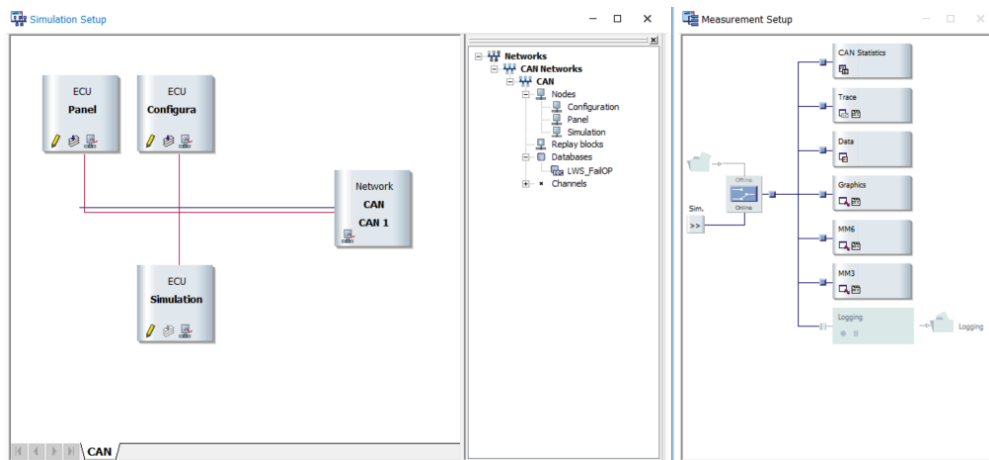


Figure 4-11: CANoe Environment

Other supported tools that are relevant for the Panels are the use of databases with messages pre-configured, this enables for faster analysis of the recognized frames. Other powerful tool is the ability to plot graphs while data is being received, this is great to understand how signals are progressing in time.

4.3.2 CANoe panels use cases

This section will describe the different and varied use cases that each panel has. Each of the four panels has different objectives and core functions, however they all provide enhancements or complements to the main panel and function of displaying the current angle of the steering angle and the current position with a visual representation of an actual steering wheel. Each of these panels graphical interface can be seen in Appendix A – Panels GUI.

4.3.2.1 Demonstration Panel

The demonstration panel is the main panel for the LWS Fail-Op demo, it is in this panel that is possible to access the other 3 panels. It works as a standalone panel, what it is meant is that

without any other type of configuration, the main purpose of this GUI is achieved: the visual representation of the current angle for the steering wheel. Its feature list includes start/stop the program, display the current angle and speed value, error values, visual representation of the current steering angle position, open and closing other panels, naming them from the top left to the top right are the simulation panel, the calibration panel, and the configuration panel.

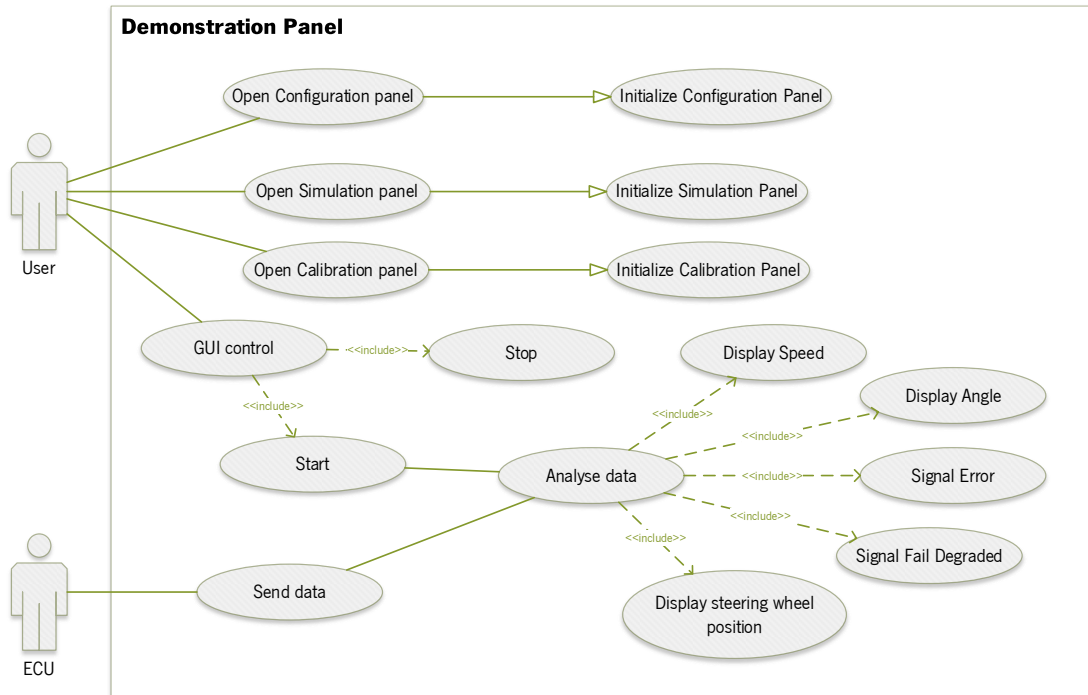


Figure 4-12: Use case demonstration panel

4.3.2.2 Simulation Panel

The simulation panel, serve the purpose as the name implies of simulating a working bus line transmitting messages to the demonstration panel. In this panel is possible to simulate most of the features.

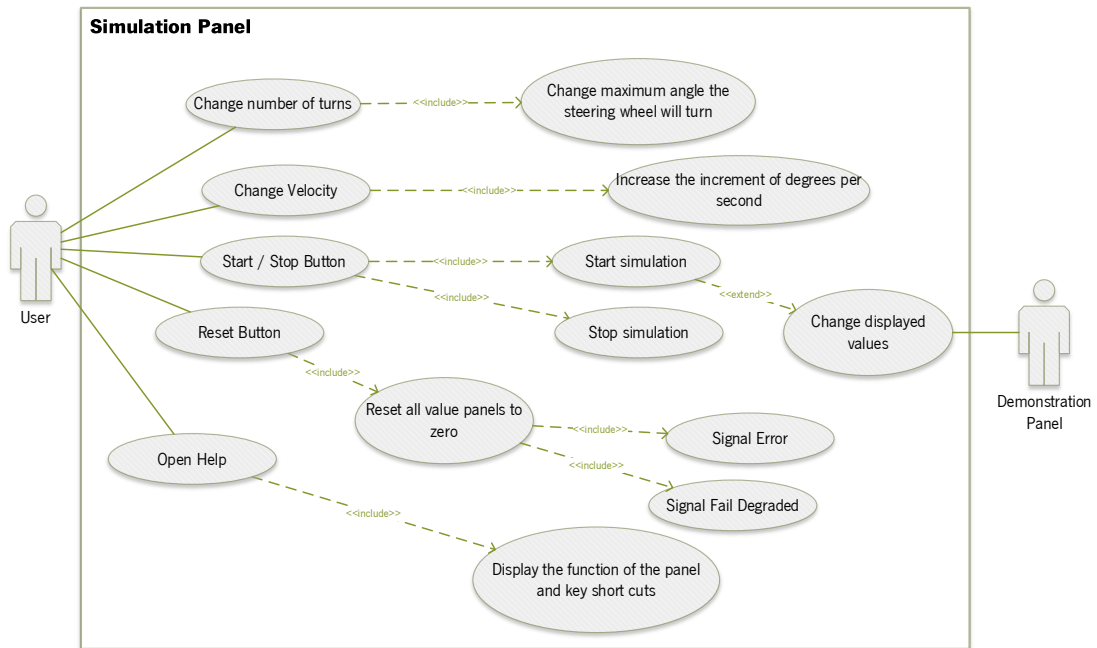


Figure 4-13: Use case simulation panel

The maximum turns represent the number of turns the steering wheel can turn either rotate to the right or left and the velocity is the number of degrees per second the steering wheel will turn. This simulation can be stopped and started at any point, as well as being reset.

4.3.2.3 Calibration Panel

The role of the calibration panel is more on the support side. It has the ability to send different types of messages that can change the way the sensor is currently working. The command of calibration allows for zero point calibration of the sensor and the command decalibration to stop the sensor sending more messages with the angle calculated. The command for MM3 and MM6 change the type of message that the sensor will be sending. One is the pure raw data and the other normalized data from the sensor. The button SW version, just waits for a returning version of the software implemented in the microcontroller. And finally, the power on and off buttons send commands to change the state of the power supply if that is supported in the system where the sensor is installed.

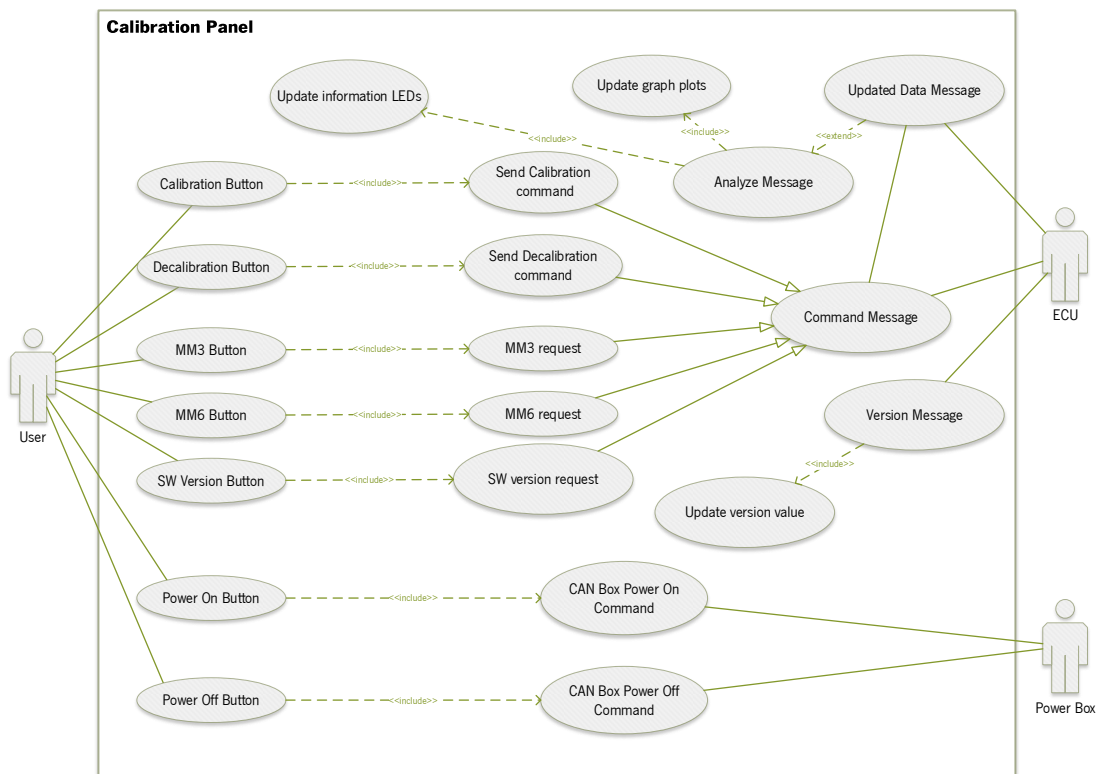


Figure 4-14: Use case calibration panel

4.3.2.4 Configuration Panel

The final panel is the configuration panel. This panel has two distinct features, the first is the ability to configure the CAN channels in while the program is running, either to be classic CAN as well CAN-FD. The other more extensive feature is the possibility to define a new type of message, this entails the ID and the type of it (by default standard 11-bit identifier), signals to be included in the frame up to 12. The size of these signals must respect the maximum size available for the current type of CAN (8 bytes for classic and 64 bytes for FD). The name allows for connecting the message signal with the demonstration panel, for example if it is named Angle it will, in fact, turn the steering angle when receiving that message even if it is not in the databases. The other configurations for the signals are the byte order to read them and the factor to multiply the received value. The final message is being shown responsively in the message layout with the colour codes attached on top.

The final three features are the ability to reset all fields, save the current message, and load a previous done message.

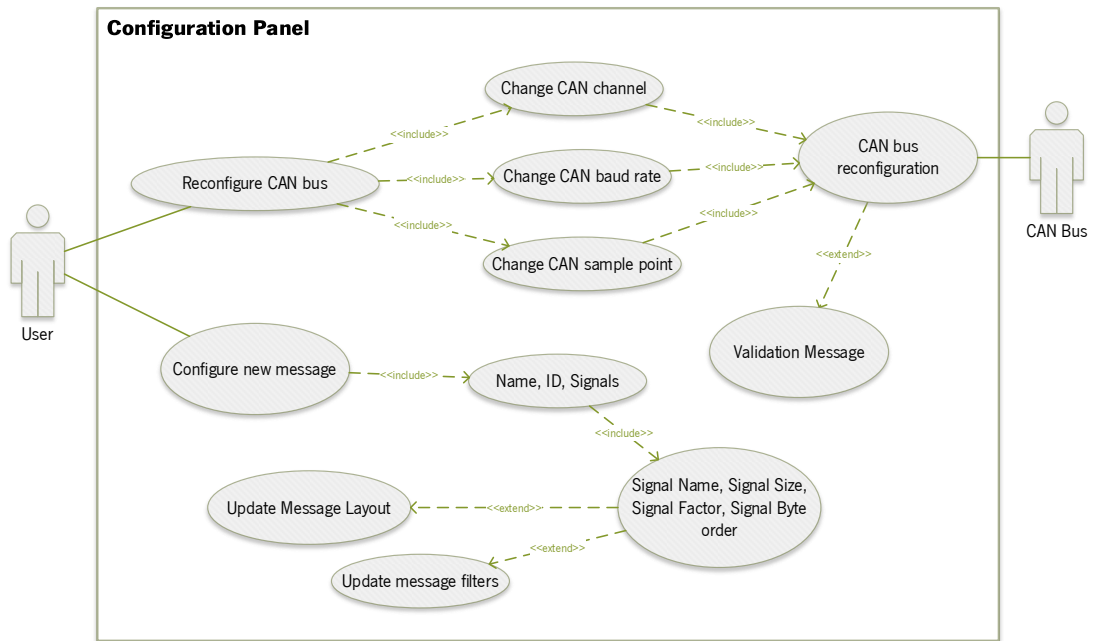


Figure 4-15: Use case configuration panel

4.4 Conclusion

This chapter described the steps taken during the development phase in order to achieve a fully functional CAN driver capable of sending and receiving messages, handle error events, and work agnostically with different CAN transceivers. All the developed code followed rules from MISRA and ISO26262, to reduce safety hazards that could be introduced by the developer. Also, the future use of an ASIL-D compiler can provide the full coverage for an ASIL-D approved code.

Chapter 5

Tests and Results

This chapter covers the validations and verification of the requirements for the implemented system. It was intended to understand if the CAN driver was correctly implemented, and all the features are working on accord with it. It was necessary to check if it works properly in both classic CAN and flexible data rates CAN, with different message sizes.

The CAN driver will be tested in both microcontrollers, the S32K116 and the S32K2TV, where the scenarios will change a bit, however, there will always be present, the microcontroller, the connection to the computer (VN1610), and the CANoe for analysis. The transceivers used are different on both boards, and some have limitations, that will be explained in further detail afterwards.

The tests sets will include testing the message system, the transceiver and non-transceiver application, the bus-off error and recovery, standby operations (standby/sleep mode and leave such state) and finalize with tests on the panels created.

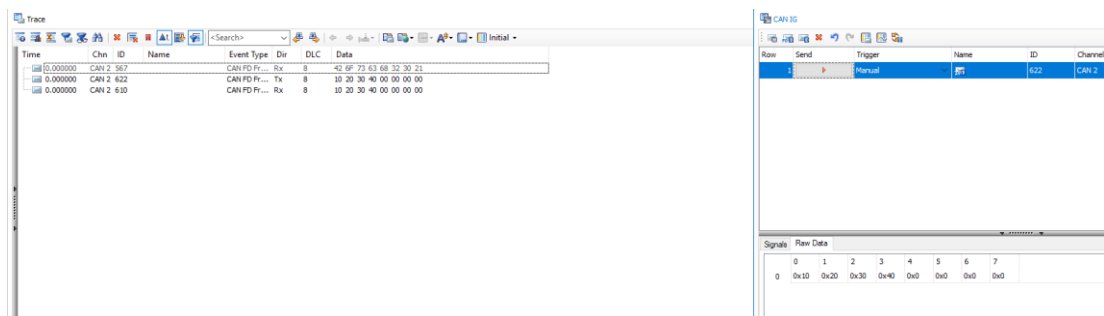
5.1 Message system validation



Figure 5-1: Test scenario for the S32K116

The message system is composed by the ability to transmit and receive messages from a microcontroller in a valid bus line, in this case the CAN bus created will be between the microcontroller and a VN1610 that will convert the data to USB so it can be read or written in the CANoe software.

The first test is the simple validation of the transmission and reception of messages. To do this, there will be a message sent from the CANoe to the microcontroller, which will respond by echoing the same data into a new ID, for easier identification. The test result is shown the Figure 5-2, the configuration is the filter on time, which shows the difference in time between the same messages and the interactive generator that enables the easy creation of a message to be sent. The baud rate and type of CAN do not affect this test, however, for record purposes it was CAN-FD at 2Mbps. Three messages were sent the first one is from the microcontroller to test the transmission of the CAN, the second one is from the CANoe and the last one is the echo from the message transmitted from the CANoe, it has a different ID just for easier identification, however the data is kept the same. Also to note, the order shown is the order that each message was received, the zero on the time indicates that only one of each was transmitted.

**Figure 5-2: Communication between the CANoe and the microcontroller**

The CAN driver can operate in a wide range of data rates. However, since the applications normally are intended from some considerable data transfers only velocities above 250 kbps up to 8 Mbps are tested. Nevertheless, slower speeds are possible. For this test the message has 8 bytes for payload and the busload is the measure to understand the velocity at which the messages are being sent. Busload is the percentage at which relevant bits are travelling on the bus, all the idle time reduces the busload, on a specific period.

Baud rate	Expected Result (us)	Result (us)
Classic: 250 kbps	452	454

Classic: 500 kbps	226	227
Classic: 1000 kbps	113	113
FD: 500kbps and 2Mbps	104	107,7
FD: 1000kbps and 2Mbps	77	79,1
FD: 500kbps and 4Mbps	79	82,7
FD: 1000kbps and 4Mbps	52	54,5
FD: 500kbps and 5Mbps	74	77,8
FD: 1000kbps and 5Mbps	47	49,2
FD: 500 kbps and 8Mbps	66.25	70,3
FD: 1000 kbps and 8Mbps	39.25	41,9

Table 5-1 - Latency of messages at different baud rates

In Table 5-1, it is possible to see the impact that the transmission rate of the arbitration phase has on the overall transmission latency, and it is worsen by the small payload, in the tests below it will be possible to see how the payload affects the efficiency. It is also possible to see that the messages are being sent with the correct baud rates to the bus, as the theoretical value corresponds to the real result.

All these values were recorded, and all the tests will be annex, the Figure 5-3 there is a difference in the average velocity because of the busload. In the first figure, it is possible to see that the busload is only around 60%, this is justified for the fact of how the driver works combined with the CAN controller. In the image on the right, it is possible to see already that the busload is near 100%. Done the math, the average message time is around the same, even if the results are not, and is more efficient to send different messages, in terms of busload.

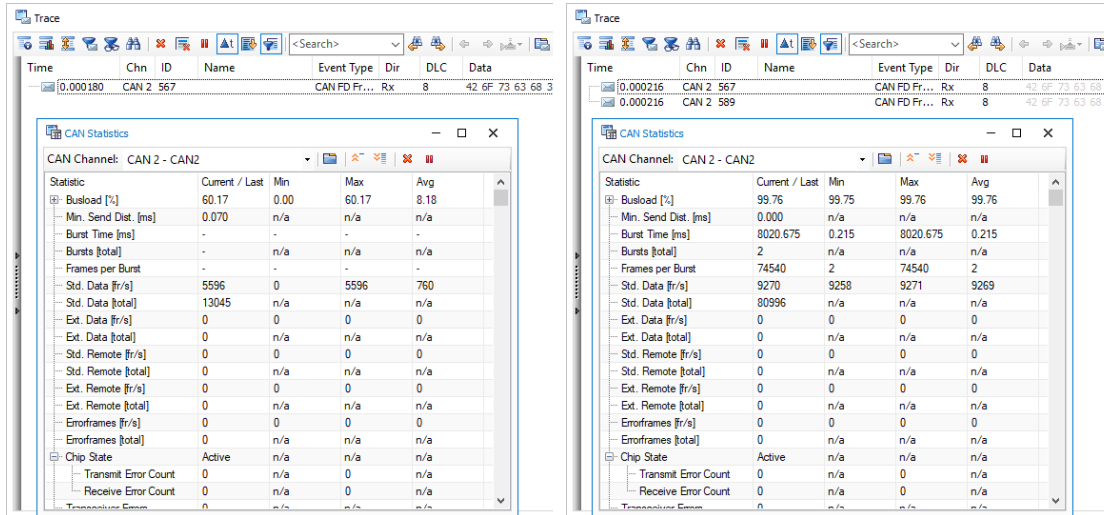


Figure 5-3 - Messages being transmitted at 2Mbps in the data phase and 500 kbps in the arbitration phase

Testing the message payloads and the overall efficiency CAN-FD was possible to get some data, where the messages ranged from 0 bytes payload up to the maximum of 64 bytes, with data carefully chosen to reduce the number of stuff bits to 0. This was achieved by just sending several 0xA, since the binary is 1010b which makes it impossible to activate the bit stuffing mechanism. All this data was organized and displayed in Table 5-2. From this table it is also possible to draw graphical representation, making it clear the increase in efficiency by sending bigger CAN-FD messages, in Figure 5-4.

Message Size (bytes)	Transmission Time CAN-FD 2Mbps (us)	Average data rate (Mbps)	Transmission Time CAN-FD 8Mbps (us)	Average data rate (Mbps)
0	47,5	1,284	34,5	1,768
4	63,5	1,465	38,6	2,416
8	79,5	1,572	42,6	2,934
12	95,5	1,644	46,6	3,369
16	115,25	1,683	49,6	3,810
32	178	1,781	67,2	4,717

64	306,5	1,869	99,5	5,759
----	-------	-------	------	-------

Table 5-2 - Average data rates for different CAN velocities

The ramp-up in the, more noticeable in the 8 Mbps transmission is due to the increase in the CRC size on messages above 16 bytes, since the CRC is still on the higher transmission rate it makes the average raise. Any message bigger than 64 bytes will have to be split into different messages, since the driver does not support chained messages as of the moment.

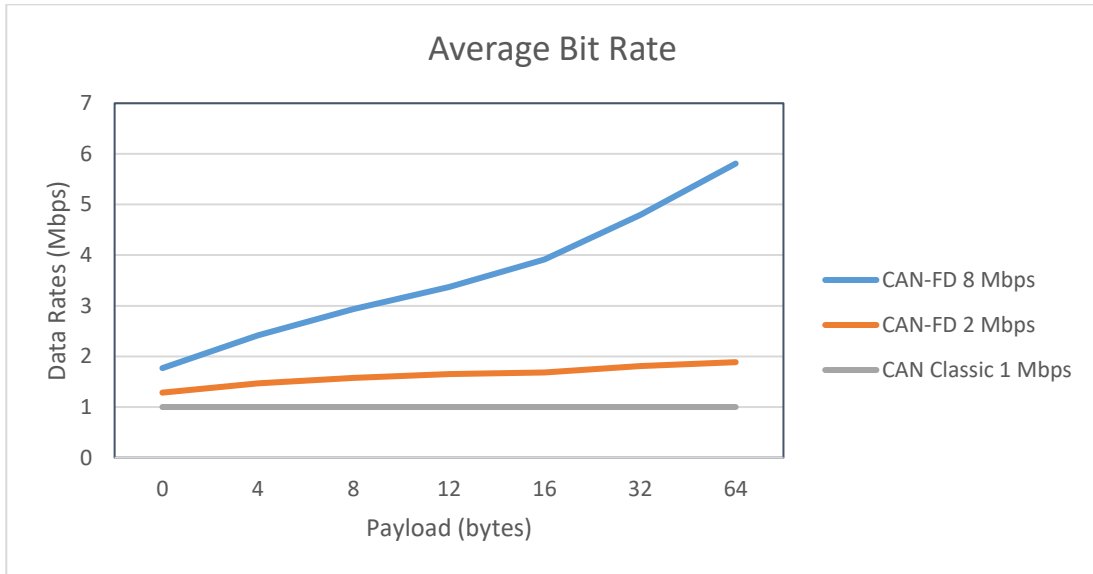


Figure 5-4 - Graphical representation of the average bit rate with different baud rates

Additionally, all the extra tests can be found in the annex part of the document, all of which will be similar to the images in Figure 5-4. The key data to take from these images is the data size (since the data in itself is on the same spectre) and the time that took to send a message.

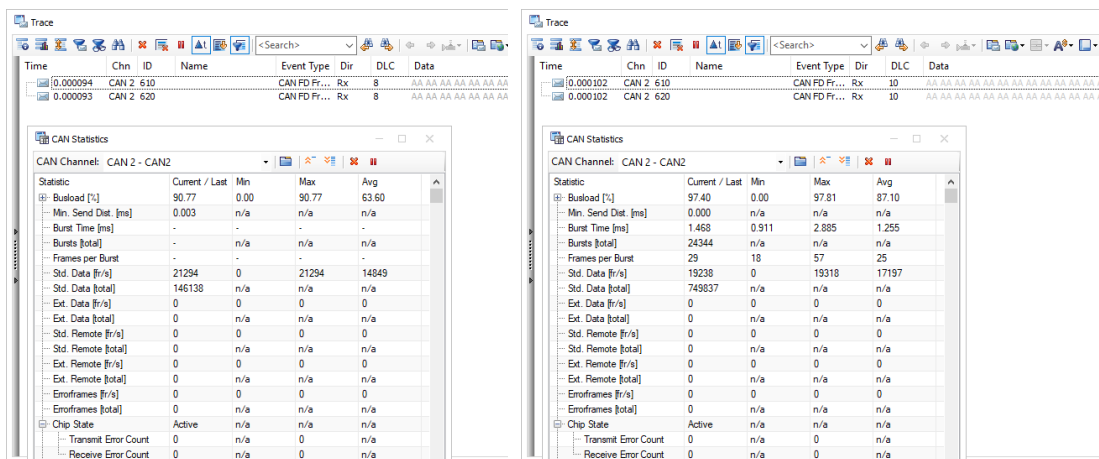


Figure 5-5 - Transmission time for 8 Mbps baud rate and 500 kbps on a 8 byte payload and 16 byte payload respectively

5.2 Bus-off event validation

The bus-off state in a microcontroller is described by its inability to communicate, but this only happens to one CAN channel/bus at a time and is data rate independent. This means that even if, one channel needs recovery the other channels may still be communicating. To enter this state there must happen an accumulation of errors, which can be easily triggered by shunting both the CAN High and Low.

That is the scenario that will be followed, first, there will be a trigger for the bus-off and the CANoe should stop receiving messages, and the program will start trying to recover the bus line failing every time until both lines are separated. As shown in Figure 5-6, until the second 27th the CANoe is receiving several messages per second, then there is a bus-off occurrence and it stops receiving messages, after a few seconds, on the 57th second to be more precise the bus-off trigger is stopped and successful recovery is completed, making it possible to receive new messages on the CANoe.

Time	Chn	ID	Name	Event Type	Dir	DLC	Data
24.272...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
24.460...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
24.647...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
24.835...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
25.022...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
25.210...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
25.398...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
25.585...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
25.773...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
25.960...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
26.902...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
27.089...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
27.277...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
57.998...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
58.186...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
58.373...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
58.561...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
58.748...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
58.936...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
59.124...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
59.311...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
59.499...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
59.686...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
59.874...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
60.061...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
60.249...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
60.437...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	
60.624...	CAN 2	567	CAN FD Fr...	Rx	8	42 6F 73 63 68 32 30 21	

Figure 5-6 - Bus-off event and subsequently recover from the bus-off

5.3 Non-transceiver validation

Validating a non-transceiver application passes through proving that the messages are being sent and acknowledged. It is also important to understand that how the physical layer is set up can very well influence the maximum transmission rates and the recovery time of the lines. To consider is also the delay added by isolation layers. Where the first affects the sample point and the second affects an added delay on feedback.

Figure 5-7, shows a Picoscope view configured into two windows. The top windows are measuring the non-transceiver communication and the bottom is monitoring both high and low CAN bus lines from the differential transceiver. The same message is sent into both communications, however since the communication without transceivers cannot be easily decoded by the CANoe, the mirror message serves as an easier visual representation as well validation of the information on the message.

The main points to decode in the message are the ID and the data, it is also interesting to see the point at which the data goes through acceleration (immediately after the BRS – bit rate switch - bit) and goes back to the arbitration velocity on the validation of the Acknowledge. Starting by the ID, it is possible to see that the bits are 10101010101b, translated to hexadecimal 0x555 which is the same that is translated in the CANoe in Figure 5-8. As for the data is 01010011b in binary as for hexadecimal is 0x53, that exactly matches the received CAN message.

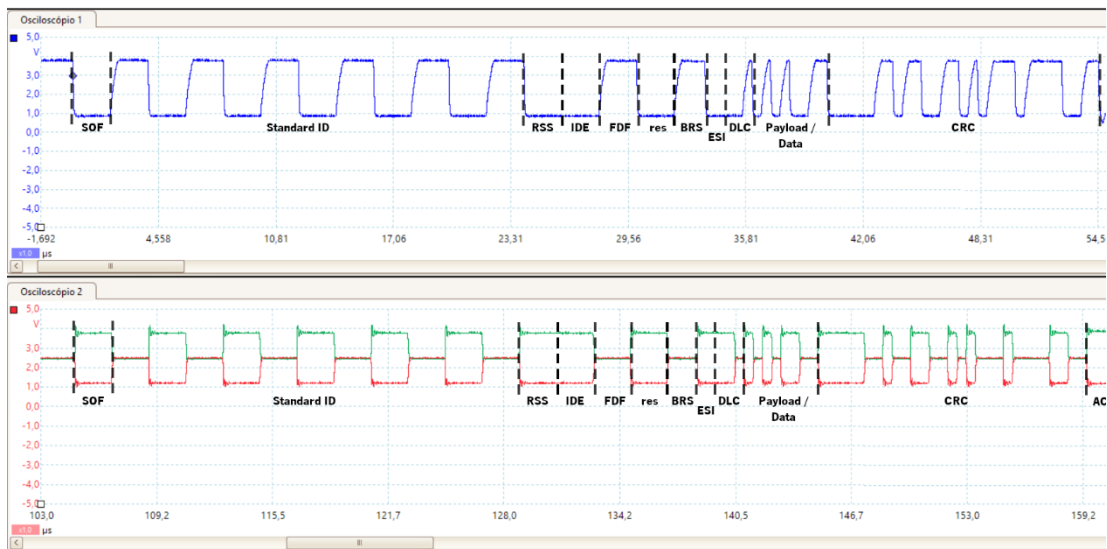


Figure 5-7 - No transceiver measure and differential CAN bus measure

Time	Chn	ID	Name	Event Type	Dir	DLC	Data
14.657295	CAN 2	555		CAN FD Fr...	Rx	1	53

Figure 5-8 - Message mirroring the message sent on the no transceiver communication

5.4 Standby operations validation

The standby entry is characterized by the fall of consumption from a development board and when it wakes the consumption rises once again until it picks once every configured peripheral is again running and the amount of load in the processor.

The test scenario will be as follows have two buttons for inputs, the first will trigger the configuration and the latter will trigger the sleep or standby mode. From the initial state, the board will initialize will the default clocks and security configurations, followed by the start of the core application, that will turn on the green light and only check for button inputs. The first button trigger runs the configuration of the CAN and the ADC, and turns the blue light on. At this point it is possible to see the maximum consumption that will be registered in this configuration and both lights on. On the second trigger, both lights are turned off and the system is shutdown, entering standby mode. Once the wake-up trigger happens the core application is resumed turning on again the green led and waits for a new configuration.



	Event	Consumption (mA)	Visual Queue
	Initial State	60	
▶	Start running	61	
👉	Configuration trigger	114	
👉	Standby trigger	52	
🔊	Interrupt and Standby leave	61	
👉	Configuration trigger	114	

Table 5-3 - Test scenario testing the standby operation

5.5 Panels validation

The LWS-Fail-OP Demo is as stated before, a group of graphical panels intended to help users visualize the working within a steering angle sensor. It supports extra configuration functions, simulation and calibration.

5.5.1 Demonstration panel

The demonstration panel is the main panel that holds the core function of the LWS Fail-Op demo, which is to visualize all the data from the steering wheel message. In Figure 5-9 is possible to see the message and the reaction of the panel. The message brings a collection of information where the value of the Steer Wheel Angle determines the current position of the steering wheel on the panel that mirrors the position of the real steering wheel.

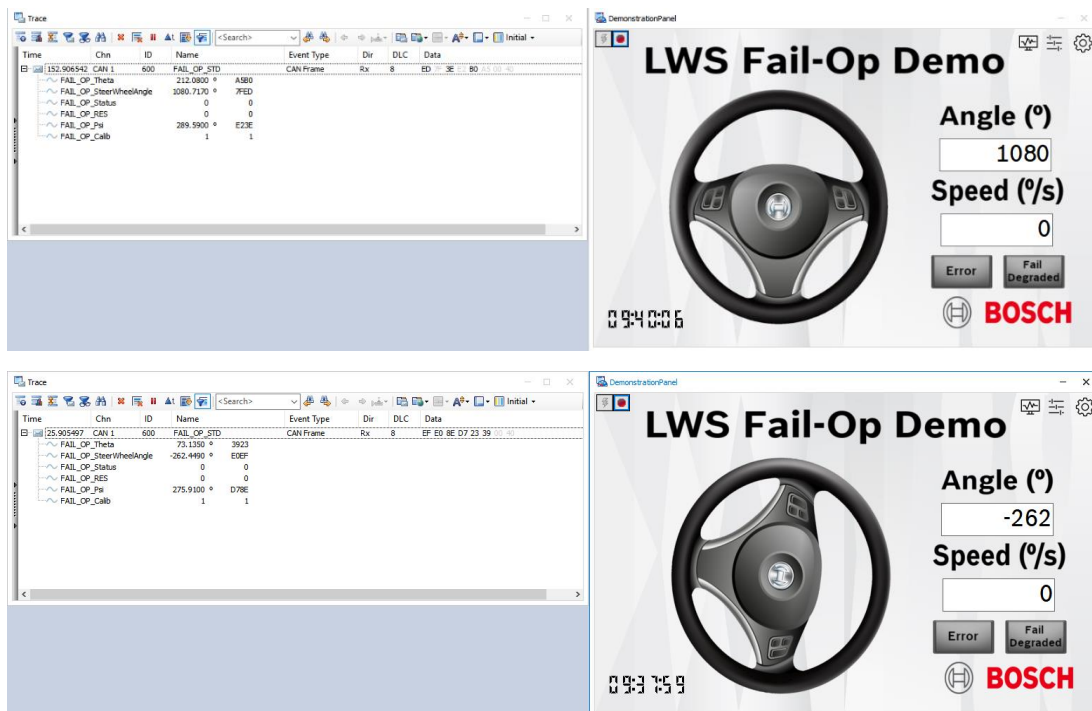


Figure 5-9 - Demonstration Panel and the analysed message

Depicting the rest of the message, there are 2 extra signals with angular values, the Psi and the Theta, that represent the angle of each of the gears. Besides that, there are 14 bits reserved for extra information in the future. And bit to indicate the status of the sensor and another to show if the sensor is calibrated or not. In case of error, the panel signals by turning on the error LED and if it enters in fail-degraded, meaning only one of the microcontrollers is transmitting, the other LED turns on, as shown in Figure 5-10. That message induces the panel to turn on both LEDs. The fail-degraded is indicated by the FAIL_OP_Status bit of the message

that is one, and the error is indicated by the fact that the angle value is 0x7FFF even though that the sensor is calibrated, which indicates an error.

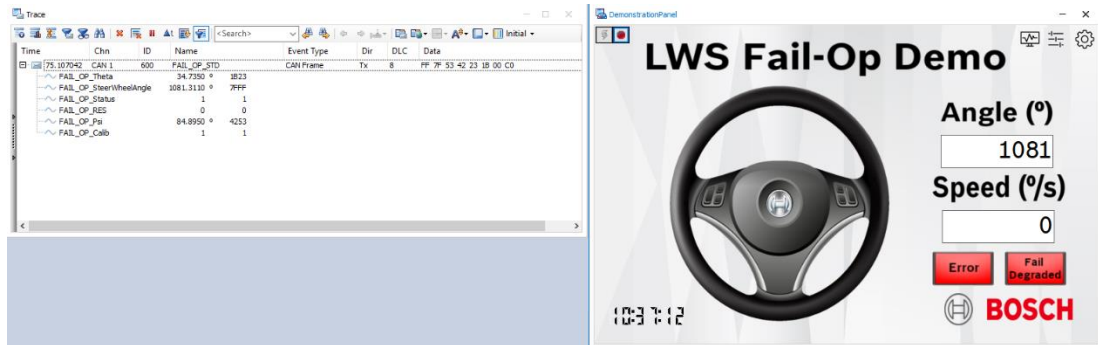


Figure 5-10 - Demonstration panel status and error indication

5.5.2 Configuration panel

The configuration panel has 2 main features: the ability to reconfigure the CAN bus on run time and configure a new message on run time capable of identifying a part of its elements to display them on the main panel.

To test the first mentioned feature, the CANoe bus line was configured to CAN-FD with 500kbps at the arbitration phase and 1 Mbps at the data phase. However, the messages being sent by the microcontroller were at 2 Mbps instead of the 1Mbps, these means that all the received messages will produce an error since the sample point will miss every time once the bit rate switch starts. This event is shown in Figure 5-11, where the first messages are received with an error and are signaled with red. After a reconfiguration signaled with a success message, the CAN is restarted with the new values and is now possible to receive the messages that are now marked with black and it is possible to decode the frame.

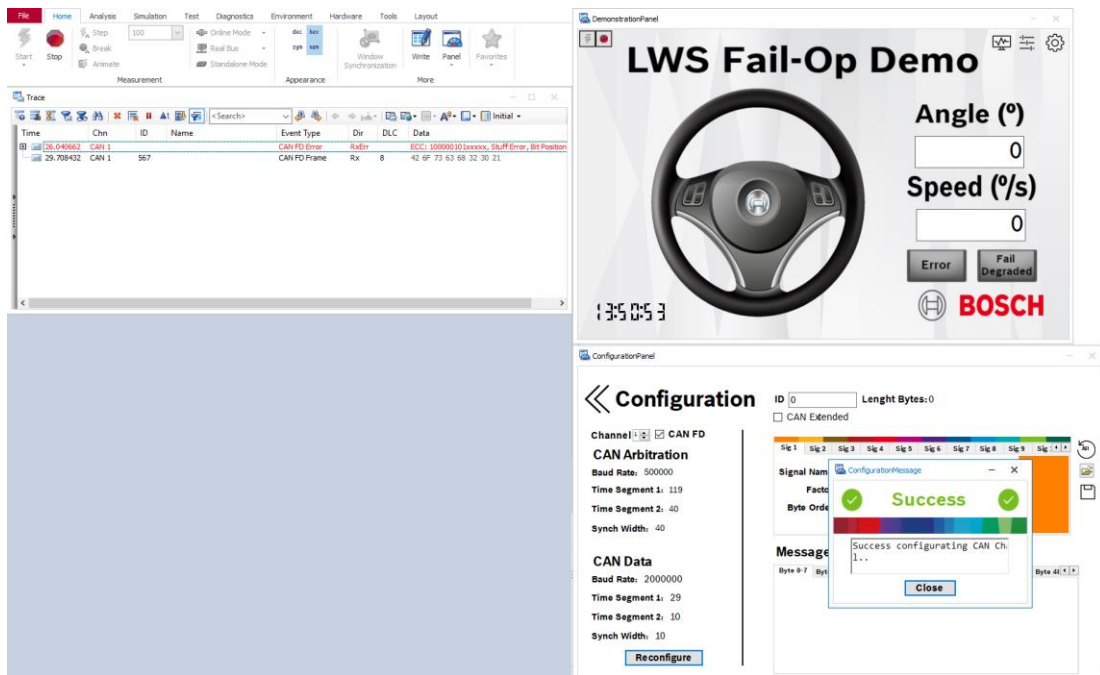


Figure 5-11 - Reconfiguration of the CAN bus line

The other function on the configuration panel is the ability to redefine a message so it can be translated. In this case the configured message has 6 different signals. Being the first one the Angle, this is important, because is the name of the signal tells where to acquire data in this case, the first 16 bits. And as expected from the first 16 bits in intel endianness, 0x73B9 and the factor of 0x033 it comes around 977° as represented in the picture. Also the ID must match with one of the received messages, in this case 0x567 in hexadecimal (1383 in decimal). The big advantage of this is being able to work in both CAN classic and CAN-FD.

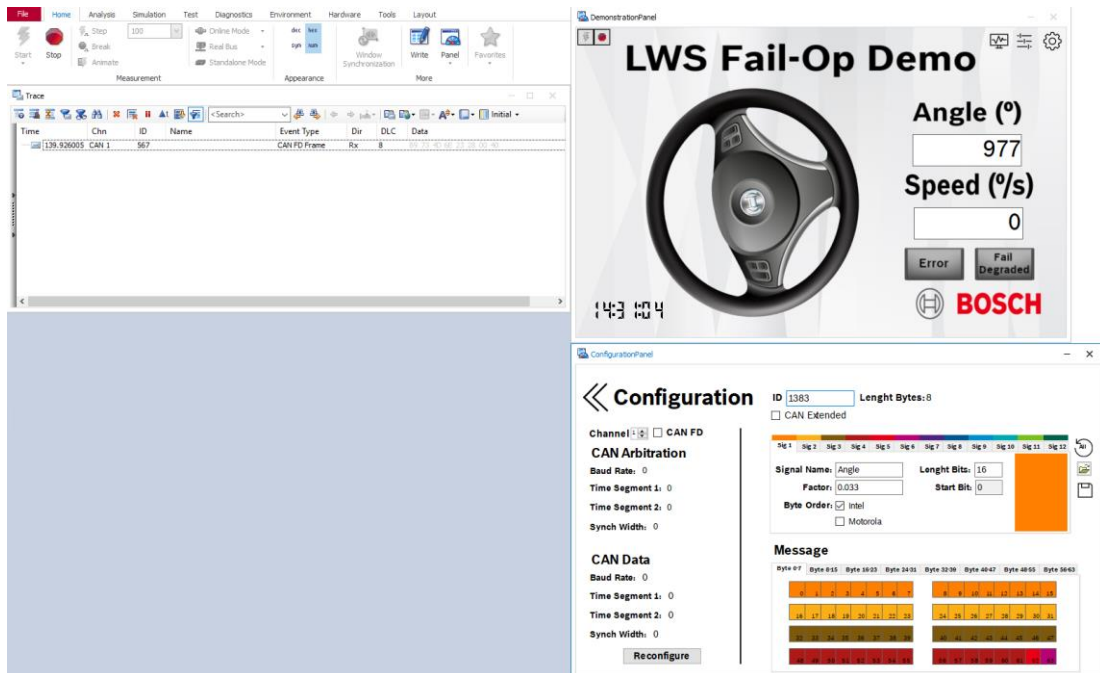


Figure 5-12 - Custom message to receive data from a new message

5.5.3 Calibration panel

The calibration panel is responsible for a series of commands that aid or support the functionalities of the main demonstration panel. Since they are only commands, they only send a message when a button is pressed and the state is changed depending on the standard message received.

The three main functions of the calibration panel are the ability to zero-point calibration or reset calibration of the sensor via a command, change the incoming messages from standard to prior calculating points, with MM6 and MM3 with raw data or the calculated gear angles respectively. And finally, the power box, this power box is a supporting tool for testing and development that allows to power cycle the board with a CAN command. The last least important feature is just the request of the current version, as shown in the Figure 5-13.

All these commands use the same configuration message ID with different data. Except for the power box which has its own ID and configuration. In Figure 5-14, is possible to see the messages for calibration and decalibration and how they affect the standard message as expected. The LED turns green when the state changes in this case from calibrated state to decalibrated state as can be also seen in the standard message Calibration bit. For the other commands it works the same way, indicating the state of the messages at any time.

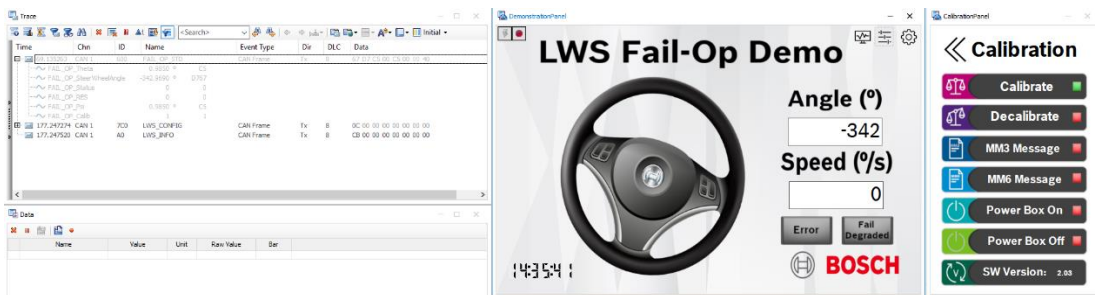
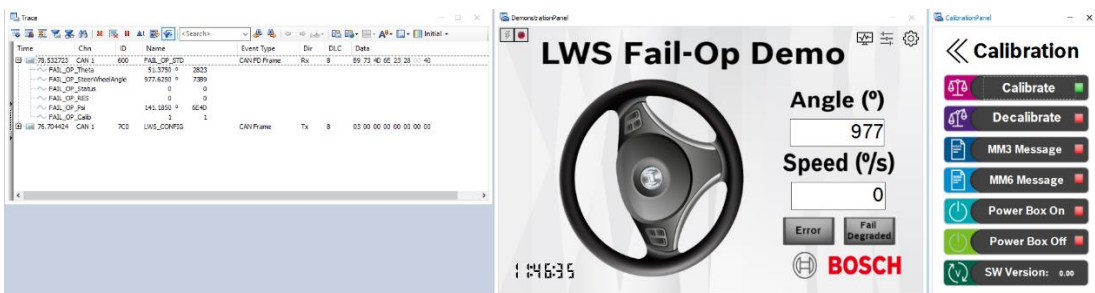


Figure 5-13 - Command requesting the software version



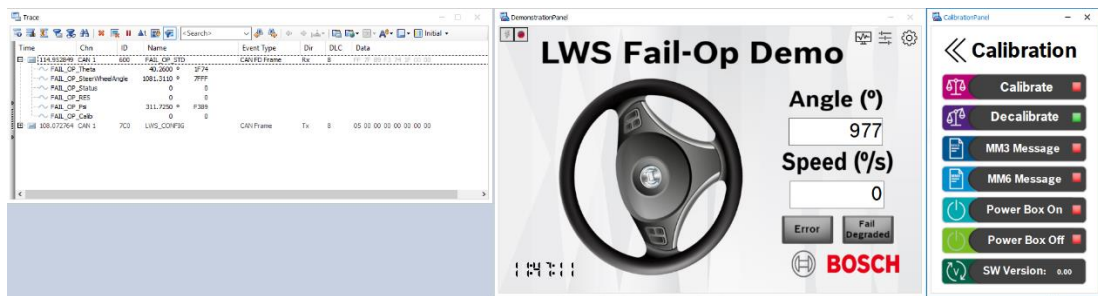


Figure 5-14 - Calibration and Decalibration of the sensor and its standard message

5.5.4 Simulation panel

There were extensive tests on all the capabilities and limits for the simulation panel, however the only relevant test on the scope of this document is to understand if the steering wheel does really turn according to the simulation. This can be hard to prove via images, nevertheless, Figure 5-15 compiles a collection of images from different stages of the simulation. From rotating it to the maximum in the defined turns, then rotating it in the opposite direction. And finally rotating it at a maximum velocity of 1080° per second. Even though in the last one, the GUI cannot keep up with the fast update rate.

This screenshot shows the initial state of the LWS Fail-Op Demo. The 'DemonstrationPanel' window displays a steering wheel icon, a digital clock at 14:37:30, and control fields for 'Angle (°)' set to 1260 and 'Speed (%/s)' set to 90. There are 'Error' and 'Fail Degraded' buttons, and the Bosch logo. The 'SimulationPanel' window shows a 'Simulation Panel' title, a color bar, 'Max Turns' set to 3.5, and a 'Velocity (°/s)' display showing 0090. A slider below the velocity display ranges from -1080 to 1080.

This screenshot shows the LWS Fail-Op Demo after a transition. The 'DemonstrationPanel' window now shows the steering wheel icon rotated, a digital clock at 14:38:49, and control fields for 'Angle (°)' set to 750 and 'Speed (%/s)' set to -45. The 'Error' button is now disabled, and the 'Fail Degraded' button is active. The 'SimulationPanel' window shows the 'Velocity (°/s)' display updated to -0045, with the slider moved to the left.

This screenshot shows the LWS Fail-Op Demo in a further state. The 'DemonstrationPanel' window shows the steering wheel icon rotated further, a digital clock at 14:39:13, and control fields for 'Angle (°)' set to -350 and 'Speed (%/s)' set to -45. The 'Error' button is now active, and the 'Fail Degraded' button is disabled. The 'SimulationPanel' window shows the 'Velocity (°/s)' display still at -0045, with the slider moved to the right.

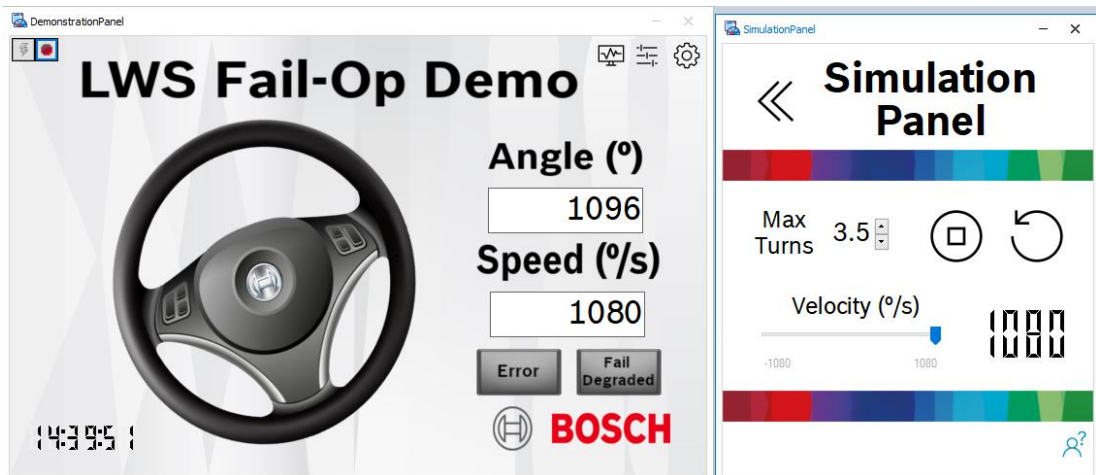


Figure 5-15 - Different stages of simulation

Chapter 6

Conclusion and Future Work

This chapter aims to summarize and provide an overview of the implemented steps and obtained results from either the development of the CAN driver as well as the GUI, and respective tests and results performed on each part. As a final note, this chapter also intends to suggest future improvements or additional features that could benefit both implementations.

6.1 Conclusion

More than ever, nowadays everything is connected, either being cars or personal objects. This communication link can be implemented by using dozens of different protocols. However, for each one of them depending on level of required safety it is important to step back and design them with different features in mind. From ASIL-D hardware to safe software practices each step is important to guarantee the availability and reliability of the protocol used.

The importance of MISRA guidelines creating a subset of C programming language with great emphasis in static checking tools to enforce such subset, and with that approach improves the robustness of the software developed, from the design, through the implementation up to the tests and validation.

Other guidelines focused more on the syntax of the written code, such as naming conventions, style guides and use of defensive implementation techniques are crucial to write a safer and clear code. With this, the code can be developed by different programmers, while keeping it readable and easier for future testing and validation. Finally, the use of the V model from the ISO 26262 for the software. This includes product development at the software level, the specification of the software requirements and its architectural design, the software implementation process, and the software verification, integration, and testing of the embedded software.

The driver developed in this dissertation was able to fulfil all the requirements with little to none deviation. From transmitting capabilities to receiving capabilities with different and flexible configurations. Different CAN transceivers are supported in implementations (even no transceivers at all), within the full high-speed range and low-speed range for the CAN classic and CAN-FD.

6.2 Future Work

The more flexible the driver the best features can be added and tested. It is crucial when designing any snippet of software to make clear cuts where a function starts and ends, fragmenting and isolate parts of the code, creates an easier platform to work on and to test on. This said, it is possible to improve and to add features into different sectors.

Improve on the network manager: The network manager is responsible for the transition from normal operation to bus-sleep operation, which is a supported feature, however this could also include a service to detect all the present nodes or even a service to detect which nodes are ready to sleep. Both these services could be implemented in a future version of the driver.

Remove the wrappers: On the application layer wrappers were created for easier interaction with the functionalities of the CAN driver, mostly transmission and reception operations. These operations were inspired in the `printf()` and `scanf()` functions. On future versions and with AUTOSAR style integrations there would be no need for the use of such wrappers and therefore they could be deleted.

Encryption: With the CAN-FD supported capabilities the data size just became exponentially bigger. This allows for the data to be potentially encrypted and therefore better protected against hackers.

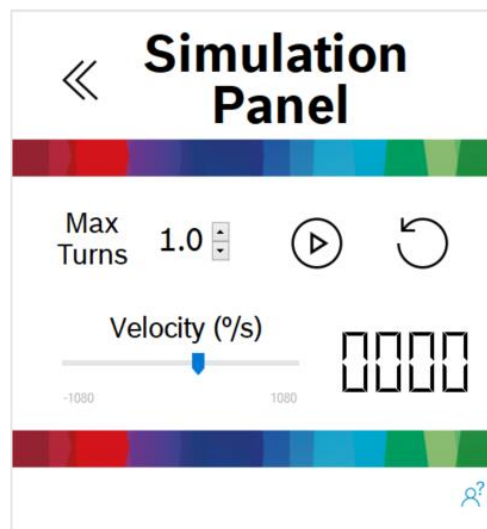
As seen in the results, the GUI is a great tool to test and operate the sensors in an easy and user-friendly way. Nonetheless, it has several points where it could be improved upon.

Support multiple CAN channels at the same time: As of now, most of the Panels only support one CAN channel at a time, only the Configuration panel was constructed with multiple channel support in mind. This means that having multiple channels, allow for different signals to be read and processed at the same time.

Support output of custom messages: On the configuration panel there is the possibility to define a message that could be received and decoded into different data to be analyzed. Using

this interface, it could also be possible to do the opposite. Instead of defining a message to be received, there could be defined one to be sent. To support this, it would also be necessary to implement methods to define data, which could be static values or simple linear functions. Other than this, support for extra CAPL files could make running automated tests and extra features possible.

Appendix A – Panels GUI



←← Calibration

-  **Calibrate** ■
-  **Decalibrate** ■
-  **MM3 Message** ■
-  **MM6 Message** ■
-  **Power On** ■
-  **Power Off** ■
-  **SW Version:** 0.00

←← Configuration

Channel CAN FD

CAN Arbitration Phase

Baud Rate: 0

Time Segment 1: 0

Time Segment 2: 0

Synch Width: 0

CAN Data Phase

Baud Rate: 0

Time Segment 1: 0

Time Segment 2: 0

Synch Width: 0

Reconfigure

ID: Length Bytes: 0

CAN Extended

Sig 1	Sig 2	Sig 3	Sig 4	Sig 5	Sig 6	Sig 7	Sig 8	Sig 9	Sig 10	Sig 11	Sig 12
Signal Name: <input type="text"/>											Lenght Bits: <input type="text" value="0"/>
Factor: <input type="text" value="1.00"/>											Start Bit: <input type="text" value="0"/>
Byte Order: <input checked="" type="checkbox"/> Intel <input type="checkbox"/> Motorola											

Message Layout

Byte 0-7	Byte 8-15	Byte 16-23	Byte 24-31	Byte 32-39	Byte 40-47	Byte 48-55	Byte 56-63

The figure displays four sequential screenshots of the LWS Fail-Op Demo GUI, illustrating the system's response to various events. Each screenshot includes a central steering wheel graphic, a Bosch logo, and real-time data for Angle (°) and Speed (°/s). The left side of each screenshot shows a trace window with CAN bus data.

Top Screenshot (Initial State):

- Angle (°): 0
- Speed (°/s): 0
- Buttons: Error, Fail Degraded
- Trace: Shows CAN messages for CAN Error and CAN Data.

Second Screenshot (Configuration):

- Configuration Panel: Shows CAN FD settings, CAN Arbitration (Baud Rate: 500000, Time Segment 1: 5, Time Segment 2: 2, Synch Width: 2), and CAN Data (Baud Rate: 2000000, Time Segment 1: 29, Time Segment 2: 10, Synch Width: 10).
- Message Box: "Success configuring CAN Ch. 1.."
- Trace: Shows a message: "12-0005 Time-Sync: Deactivating software time synchronization. (HW-Sync active for all channels or <...>".

Third Screenshot (Calibration):

- Angle (°): 977
- Speed (°/s): 0
- Buttons: Error, Fail Degraded
- Calibration Panel: Shows buttons for Calibrate, Decalibrate, MM3 Message, MM6 Message, Power Box On, Power Box Off, and SW Version: 0.00.
- Trace: Shows CAN messages for FAIL_OP_Theta (ID: 34, Data: 31370), FAIL_OP_SteerWheelAngle (ID: 1081, Data: 3FFF), FAIL_OP_Status (ID: 1, Data: 1), FAIL_OP_RES (ID: 0, Data: 0), FAIL_OP_Psi (ID: 84, Data: 8950), and FAIL_OP_Calb (ID: 1, Data: 1).

Bottom Screenshot (Failure State):

- Angle (°): 1081
- Speed (°/s): 0
- Buttons: Error, Fail Degraded
- Trace: Shows CAN messages for FAIL_OP_STD (ID: 600, Data: 347350), FAIL_OP_Theta (ID: 34, Data: 31370), FAIL_OP_SteerWheelAngle (ID: 1081, Data: 3FFF), FAIL_OP_Status (ID: 1, Data: 1), FAIL_OP_RES (ID: 0, Data: 0), FAIL_OP_Psi (ID: 84, Data: 8950), and FAIL_OP_Calb (ID: 1, Data: 1).

References

- [1] S. Purnendu, "Architectural design and reliability analysis of a fail-operational brake-by-wire system from ISO 26262 perspectives," *Reliability Engineering & System Safety*, vol. 96, pp. 1349-1359, October 2011.
- [2] S. B. Balajee, P. Balaji, J. Shreyas and K. Satheesh, "An Overview of X-By Wire Systems," *First National Conference on Emerging Trends in Automotive Technology (ETAT-2015)*, 2011.
- [3] "ISO 26262-3 Road vehicles – Functional safety – Part 3: Concept phase," 2018.
- [4] A. Kader, "Steer-by-Wire Control System," *Swarthmore College Department of Engineering*, May 2006.
- [5] B. Chen, "practices and Challenges for Achieving Functional Safety of Modern Automotive SoCs," 2019.
- [6] S. R. e. al, "Extensibility in automotive security: Current practice and challenges: Invited," em *Proc. 54th Ann. Design Autom. Conf.*, New York, 2017.
- [7] "ISO 26262-4 Road vehicles – Functional safety – Part 4: Product development at the system level," Standard International, 2018.
- [8] T. Nolte, H. Hansson e L. L. Bello, "Automotive Communications-Past, Current and Future," em *IEEE Conference on Emerging Technologies and Factory Automation*, Catania, 2005.
- [9] A. Gargantini, E. Riccobene e P. Scandurra, "Model-driven design and asm-based validation of embedded systems," *Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation*, pp. 24-54, 2009.
- [10] M. Barr, "Terms Starting with E," Barr Group, [Online]. Available: <https://barrgroup.com/embedded-systems/glossary-e>. [Acedido em 2 January 2020].
- [11] M. Barr, "Real men program in C," 1 August 2009. [Online]. Available: <https://www.embedded.com/real-men-program-in-c/>. [Acedido em 2 January 2020].
- [12] S. Heath, *Embedded Systems Design*, Oxford : EDN series for design engineers, 2003.

- [13] W. Honhxing e W. Tianmiao, "Curriculum of Embedded System for Software Colleges," em *2nd IEEE/ASME International Conference on Mechatronics and Embedded Systems and Applications*, Beijing, 2006.
- [14] T. Noergaard, *Embedded Systems Architecture - A Comprehensive Guide for Engineers and Programmers*, 2005.
- [15] W. Reed, "Safety critical software in traffic control systems," em *IEE Colloquium on Safety Critical Software in Vehicle and Traffic Control*, London, 2002.
- [16] J. Millward, "System architectures for safety critical automotive applications," em *IEE Colloquium on Safety Critical Software in Vehicle and Traffic Control*, London, 1990.
- [17] D. Reinhardt, D. Adam, E. Lubbers, R. Amarnath, R. Schneider, S. Gansel, S. Schnitzer, C. Herber, T. Sandmann, H. U. Michel, D. Kaule, D. Olkun, M. Rehm, J. Harnisch, A. Richter, S. Baehr, O. Sander, J. Becker, U. Baumgarten e Theiling, "Embedded Virtualization Approaches for Ensuring Safety and Security within E/E Automotive Systems," em *Embedded World Conference*, Nürnberg, 2015.
- [18] Farlex, "safety-critical system," The Free Dictionary, [Online]. Available: <https://encyclopedia2.thefreedictionary.com/safety-critical+system>. [Acedido em 2 January 2020].
- [19] I. Sommerville, "Systems, Software and Technology," [Online]. Available: <https://iansommerville.com/systems-software-and-technology/static/courses/critical-systems-engineering/>. [Acedido em 2 January 2020].
- [20] J. C. Knight, "Safety critical systems: challenges and directions," em *Proceedings of the 24th International Conference on Software Engineering*, Orlando, 2005.
- [21] J. Gray, "Why Do Computers Stop and What Can Be Done About It?," Tandem Computers, Cupertino, 1985.
- [22] F. Afonso, "Operating system fault tolerance support for real-time embedded applications," 2009.
- [23] P. Bhansali, "Perspectives on safety-critical software," em *Proceedings of Australian Software Engineering Conference ASWEC 97*, Sydney, 1997.

- [24] P. Kafka, "The Automotive Standard ISO 26262, the innovative driver for enhanced safety assessment & technology for motor cars," em *2012 International Symposium on Safety Science and Technology*, Feffernitz, 2012.
- [25] NI, "What does the functional safety standard, ISO 26262 contain?," 2020 NATIONAL INSTRUMENTS CORP., 2019 March 5. [Online]. Available: <https://www.ni.com/de-de/innovations/white-papers/11/what-is-the-iso-26262-functional-safety-standard-.html#toc2>. [Acedido em 5 January 2020].
- [26] ISO, *ISO 26262: Road vehicles - Functional Safety*, 2018.
- [27] "ISO 26262-1 Road vehicles – Functional safety – Part 1: Vocabulary," Standard International, 2018.
- [28] "ISO 26262-2 Road vehicles – Functional safety – Part 2: Management of Functional Safety," Standard International, 2018.
- [29] "ISO 26262-6 Road vehicles – Functional safety – Part 6: Product development at the software level," Standard International, 2018.
- [30] "ISO 26262-9 Road vehicles – Functional safety – Part 9: Automotive safety integrity level (ASIL)-oriented and safety-oriented analyses," Standard International, 2018.
- [31] "ISO 26262-10 Road vehicles – Functional safety – Part 10: Guidelines on ISO 26262," Standard International, 2018.
- [32] J. R. Barker, "Army test experts identify five missteps in requirements development," U.S. Army Test and Evaluation Command and Don Sando, 9 January 2020. [Online]. Available: https://www.army.mil/article/231523/army_test_experts_identify_five_missteps_in_requirements_development. [Acedido em 24 September 2020].
- [33] Perforce Software, "A guide to coding standards MISRA C and MISRA C++," [Online]. Available: <https://www.perforce.com/resources/qac/misra-c-cpp>. [Acedido em 25 September 2020].
- [34] HORIBA MIRA Limited, *MISRA Compliance:2020 Achieving compliance with MISRA Coding Guidelines*, Warwickshire: British Library Cataloguing, 2020.
- [35] AUTOSAR GbR, *Specification of C Implementation Rules*, 2008.
- [36] L. J. Rodríguez-Aragón, *Tema 4: Internet y Teleinformática*, Universidad Rey Juan Carlos.

- [37] A. Neumann, M. J. Mytych, D. Wesemann, L. Wisniewski e J. Jarperneite, "Approaches for in-vehicle communication - an analysis and outlook," em *International Conference on Computer Networks*, Lemgo, 2017.
- [38] C. Eletronics, "LIN Bus Explained - A Simple Intro," 2020. [Online]. Available: <https://www.csselectronics.com/screen/page/lin-bus-protocol-intro-basics/language/en>. [Acedido em 30 September 2020].
- [39] "Interfacing LIN with SLIC," NXP semiconductors.
- [40] "LIN (LOCAL INTERCONNECT NETWORK) SOLUTIONS," Microcontroller Division Applications, 2002.
- [41] Y. Xu, C. Zhao, X. Chen, W. Deng e J.-G. Chung, "Integrated Protocol-Operation-Controller Design based on FlexRay Communication Protocol," *Chonbuk National University*, 2012.
- [42] S. Lorenz, "The FlexRay Electrical Physical Layer Evolution," *AUTOMOTIVE 2010*, 2010.
- [43] NXP, "How FlexRay™ Works," [Online]. Available: https://www.nxp.com/files-static/abstract/overview_applications/FRWORKS.html. [Acedido em 7 January 2020].
- [44] F. Luo, Z. Chen, J. Chen e Z. Sun, "Research on FlexRay communication system," em *2008 IEEE Vehicle Power and Propulsion Conference*, Harbin, 2008.
- [45] G. Schickhuber e O. McCarthy, "Distributed fieldbus and control network systems," *Computing & Control Engineering*, vol. 8, n° 1, pp. 21-32, 1997.
- [46] B. P. U. Middletown, D. C. B. Vernon, A. G. D. Glastonbury e S. A. H. Farmington, "Message Routing in Control Area Network (CAN) Protocol". United States of America Patente 5,854,454, 29 December 1998.
- [47] "Road vehicles - Controller area network (CAN) - Part 1: Data link layer and physical signalling," Standard International, Geneva, 2015.
- [48] "Road vehicles - Controller area network (CAN) - Part 2: High-speed medium access unit," International Standard, Geneva, 2016.
- [49] "Road vehicles - Controller area network (CAN) - Part 3: Low-speed, fault-tolerant, medium-dependent interface," International Standard, Geneva, 2006.
- [50] "Introduction to CAN | Learning Module CAN," Vector Informatik GmbH, [Online]. Available: <https://elearning.vector.com/mod/page/view.php?id=333>. [Acedido em 8 January 2020].

- [51] C. Huang, *Fault Tolerant Steer-by-Wire Systems: Impact on Vehicle Safety*, University of Wollongong Thesis Collection, 2018.
- [52] A. Balachandran, *Applications of Force Feedback Steering for Steer-By-Wire Vehicles with Active Steering*, Stanford: Department of mechanical engineering of Stanford University, 2015.
- [53] M. Bertoluzzo, G. Buja e R. Menis, "Control schemes for steer-by-wire systems," IEEE Industrial Electronics Magazine, 2007.
- [54] A. Muir, "How a car works," [Online]. Available: <https://www.howacarworks.com/illustration/118/the-rack-and-pinion-system.png>. [Acedido em 2 January 2020].
- [55] Lilydale Motors, "Understanding The Difference Between Hydarulic & Electric Power Steering," 30 November 2017. [Online]. Available: <https://medium.com/@lilydalemotorsau/understanding-the-difference-between-hydarulic-electric-power-steering-4e3d29d01b30>. [Acedido em 2 January 2020].
- [56] NXP Semiconductors, "S32K1xx Series Reference Manual," pp. 1712-1719, 1749-1756, 2018.
- [57] Keil, "CAN Primer: Creating Your Own Network," Arm Ltd., 2012.
- [58] D. J. Barrenscheen, "On-Board Communication via CAN without Transceiver," SIEMENS, 1996.
- [59] NXP Semiconductores, "S32 Design Studio IDE," 2018. [Online]. Available: <https://www.nxp.com/design/software/development-software/s32-design-studio-ide:S32-DESIGN-STUDIO-IDE>. [Acedido em 27 August 2020].
- [60] LDRA, "Protecting Embedded Systems with New MISRA C Guidelines," 2 May 2017. [Online]. Available: <https://ldra.com/protecting-embedded-systems-new-misra-c-guidelines/>. [Acedido em 27 August 2020].
- [61] Vector Informatik GmbH, "Testing ECUs and Networks with CANoe," 16 July 2020. [Online]. Available: https://www.vector.com/int/en/products/products-a-z/software/canoe/?gclid=EAlaIQobChMI5YCQ_5a76wIVENd3Ch3Uog3oEAYASAAEgl4bvD_BwE. [Acedido em 27 August 2020].
- [62] D. R. Wright, "Finite State Machines," *CSC216*, p. 28, 2005.

- [63] International Standard, "Road vehicles — Controller area (CAN) — Part 2: High-speed medium access unit," ISO, Geneva, 2016.
- [64] M. B. a. P. O. R. Reilly, Programming Embedded Systems in C and C++, 1999.
- [65] d. Annie goleman, R. boyatzis e Mckee, "Operating Systems," em *Operating Systems*, 2019.
- [66] A. M. Lister, Fundamentals of operating systems, 2013.
- [67] F. R. A. B. D. O. F. L. L. O. R. R. Gennaro S. Rodrigues, "Analyzing the Impact of Fault-Tolerance Methods in ARM Processors under Soft Errors Running Linux and Parallelization APIs," em *IEEE Transactions on Nuclear Science*, 2017.
- [68] P. A. L. a. J. F. DeFranco, "Software engineering of safety-critical systems: Themes from practitioners," em *IEEE Transactions on Reliability*, 2017.
- [69] J. Hatcliff, "Certifiably safe software-dependent systems: challenges and directions," em *Proceedings of the on Future of Software Engineering*, 2014.
- [70] M. Vuori, "Agile development of safety-critical software," em *Tampere University of Technology*, 2011..
- [71] B. M., "Fault-Tolerant Platforms for Automotive Safety-Critical Applications," 2003.
- [72] NXP, *S32K2TV Datasheet*, Abstatt: NXP Semiconductors, 2019.
- [73] A. Schnellbach, *Fail-operational automotive systems*, Graz, 2016.
- [74] C. Z. L. H. X. W. Y. D. Chengwei Tian, "Fault tolerant control method for steer-by-wire system," em *International Conference on Mechatronics and Automation*, Changchun, 2009 .
- [75] C. Z. L. H. X. W. Y. D. Chengwei Tiang, *Fault tolerant control method for steer-by-wire system*, Changchun: International Conference on Mechatronics and Automation, 2009.
- [76] N. N. Y.-Q. S. F. S.-L. Cédric Wilwert, *Design of automotive X-by-Wire systems*, 2007.
- [77] M. Grusin, "Serial Peripheral Interface (SPI)," SparkFun Electronics, [Online]. Available: <https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all>. [Acedido em 8 January 2020].
- [78] SFUPTOWNMAKER, "I2C," SparkFun Electronics, [Online]. Available: <https://learn.sparkfun.com/tutorials/i2c/all>. [Acedido em 8 January 2020].
- [79] S. Campbell, "Basic of UART communication," Circuit Basics, [Online]. Available: <https://www.circuitbasics.com/basics-uart-communication/>. [Acedido em 8 January 2020].