BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

DEMONSTRATIVE PLATFORM OF COMMUNICATION PROTOCOLS USED IN THE AUTOMOTIVE INDUSTRY

DEMONSTRAČNÍ PLATFORMA KOMUNIKAČNÍCH PROTOKOLŮ POUŽÍVANÝCH V AUTOMOBILOVÉM PRŮMYSLU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **TEREZA KUBINCOVÁ**
AUTOR PRÁCE

SUPERVISOR **Ing. VÁCLAV ŠIMEK**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

161513

Institut:	Department of Computer Systems (DCSY)
Student:	**Kubincová Tereza**
Programme:	Information Technology
Title:	**Demonstrative Platform of Communication Protocols Used in the Automotive Industry**
Category:	Embedded Systems
Academic year:	2024/25

Assignment:

1. Prepare a short survey of the protocols commonly used for communication in the automotive industry. Focus mainly on CAN, LIN, SPI, and I2C.
2. Study technical documentation of the selected NXP components specified by the supervisor that incorporate the protocols being analyzed in point 1) of the assignment.
3. Familiarize yourself with the practical usage of components from point 2) in combination with the S32 automotive processing platform and NXP Real-Time drivers.
4. Following the results of points 2) and 3), create a prototype of the platform based on modules with the selected components demonstrating their purpose and mutual interaction.
5. Choose an appropriate car model for the installation of the platform. Focus on suitable means to visualize data exchange between its modules.
6. Showcase safety features incorporated in the selected components with an emphasis on the detection of communication loss and eventual recovery scenarios.
7. Evaluate the achieved results and propose further directions for the development.

Literature:

- According to the instructions of the supervisor.

Requirements for the semestral defence:
Fulfillment of points 1 to 4 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Šimek Václav, Ing.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	1.11.2024
Submission deadline:	14.5.2025
Approval date:	31.10.2024

Abstract

Modern cars are a marvel of engineering, embodying a complex network of electronic components that work in harmony to deliver a seamless driving experience. At the core of this interconnected system lies a variety of communication protocols, enabling different control units to exchange data and coordinate their functions. While a lot of these lower level details go unnoticed to the everyday user, it is important to be familiar with the existence of these protocols. This thesis aims to bridge the gap between the view of a non-technical oriented audience and the lower level details of a complex car system, using a physical model to illustrate the operations behind.

Abstrakt

Moderné automobily sú výsledkom pokročilého inžinierstva a predstavujú zložitú sieť elektronických komponentov, ktoré navzájom spolupracujú, aby zabezpečili plynulý a spoľahlivý zážitok z jazdy. V centre tohto prepojeného systému sa nachádzajú rôzne komunikačné protokoly, ktoré umožňujú riadiacim jednotkám vymieňať si dáta a koordinovať svoju činnosť. Hoci tieto technické detaily ostávajú bežnému používateľovi väčšinou skryté, ich existencia je kľúčová pre pochopenie fungovania moderných vozidiel. Cieľom tejto práce je priblížiť tieto princípy aj netechnickému publiku a pomocou fyzického modelu názorne ukázať, ako jednotlivé časti systému spolu komunikujú a fungujú.

Keywords

CAN, LIN, SPI, I2C, AUTOSAR

Kľúčové slová

CAN, LIN, SPI, I2C, AUTOSAR

Reference

KUBINCOVÁ, Tereza. *Demonstrative platform of communication protocols used in the automotive industry.* Brno, 2025. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Václav Šimek

Demonstrative platform of communication protocols used in the automotive industry

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Václav Šimek. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Tereza Kubincová
May 12, 2025

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Ing. Václav Šimek and Ing. Viktor Obr, for their valuable guidance, support, and insightful feedback throughout the development of this thesis. I would also like to thank my family for always standing by me and supporting me not only during this project, but throughout my entire studies.

Contents

List of Figures

Chapter 1

Introduction

Modern cars are a marvel of engineering, embodying a complex network of electronic components that work in harmony to deliver a seamless driving experience. At the core of this interconnected system lies a variety of communication protocols, such as CAN (Controller Area Network) and LIN (Local Interconnect Network), enabling different control units to exchange data and coordinate essential vehicle functions. While these protocols are fundamental to vehicle design, diagnostics, and operation, they often remain hidden from the everyday user and can be challenging to grasp without practical exposure. Many educational tools focus on theoretical or software-based learning, which can be abstract and hard for students or new engineers to visualize.

This gap in practical, interactive learning presents an opportunity for a more engaging approach: a physical model designed to simulate the operation of these protocols in a real-world context. This project aims to bridge that gap by creating a learning platform—a physical model of a car that will serve as a teaching tool to familiarize users with automotive communication protocols. This model will simulate key subsystems within a vehicle and demonstrate how different ECUs (Electronic Control Units) communicate using these protocols. By interacting with the model, users will gain a deeper understanding of how modern vehicles operate, how data is exchanged between systems, and the practical applications of these communication technologies.

This work is structured into several chapters, each focusing on a specific aspect of the project. Chapter 2 provides an in-depth overview of fundamental automotive communication protocols, software design tools, and other relevant theoretical concepts that form the foundation of this project. Chapter 3 introduces the proposed solution, describing its concept, design approach, and intended functionality. Chapter 4 explores the technical intricacies of the implementation process, covering software and hardware architecture and the physical construction of the system described in the previous chapter. Finally, Chapter 5 presents a comprehensive evaluation of the final product and potential limitations while also suggesting possible improvements and directions for future development.

The primary objective of this project is to create an engaging and educational resource that makes the fundamental principles of automotive communication systems accessible to a broader audience. By demystifying these complex systems, the model aims to promote a deeper understanding of the technologies that underpin modern automotive engineering, making them more relatable and easier to grasp for non-specialist learners.

Chapter 2

Related Standards and Protocols

Modern automotive systems rely on a network of standardized communication protocols to enable seamless interaction between various components and subsystems. These protocols facilitate data exchange between Electronic Control Units (ECUs), which are responsible for managing critical functions such as engine performance, safety mechanisms, and driver assistance features. As vehicles become increasingly complex, standardized communication frameworks such as Controller Area Network (CAN), Local Interconnect Network (LIN), Ethernet and FlexRay have become essential in ensuring reliability, efficiency, and real-time responsiveness. These protocols not only enhance the integration of mechanical, electrical, and electronic systems but also support advancements in safety, performance, and in-vehicle entertainment. By establishing a structured method of communication, automotive standards and protocols play a crucial role in the development of modern vehicles, ensuring interoperability, security, and regulatory compliance.

2.1 Communication Protocols

A communication protocol is a set of formal rules that define how to transmit or exchange data. It ensures that the information sent and received is understood correctly by all parties involved. Communication protocols like SPI, I2C, CAN, LIN, Ethernet or FlexRay play an essential role in ensuring that the data shared between nodes is fast, reliable, secure, and real-time. As vehicles become increasingly connected and autonomous, the complexity and importance of these communication protocols will continue to grow, requiring robust, scalable, and secure information exchange strategies. There are many properties of a transmission that a protocol can define. For example, properties addressed with protocols may include:

- Frame size
- Frame structure
- Transmission speed
- Error detection
- Acknowledgment processes
- Flow control and timing
- Addressing

2.1.1 SPI

SPI (Serial Peripheral Interface) is a serial synchronous communication protocol originally developed by Motorola [14]. It is used for short-distance data transfer between microcontrollers and peripheral devices like sensors, memory chips or displays. It offers full-duplex communication[1], and although no maximum or minimum data rate is specified, it is quite common for SPI controllers to operate at speeds of at least 50MHz [1]. It operates on the master-slave paradigm, meaning the bus contains one central device (the master) which initiates communication with multiple slave devices. It is hard to find a formal separate "specification" of the SPI bus, for a detailed official description, one has to read the microcontroller data sheets and associated application notes. The SPI protocol does not include flow control, data acknowledgement or error checking, so the master has no way of knowing whether data was accurately sent and received, in fact, it has no idea if there are any slaves listening on the bus at all.

Physical Implementation

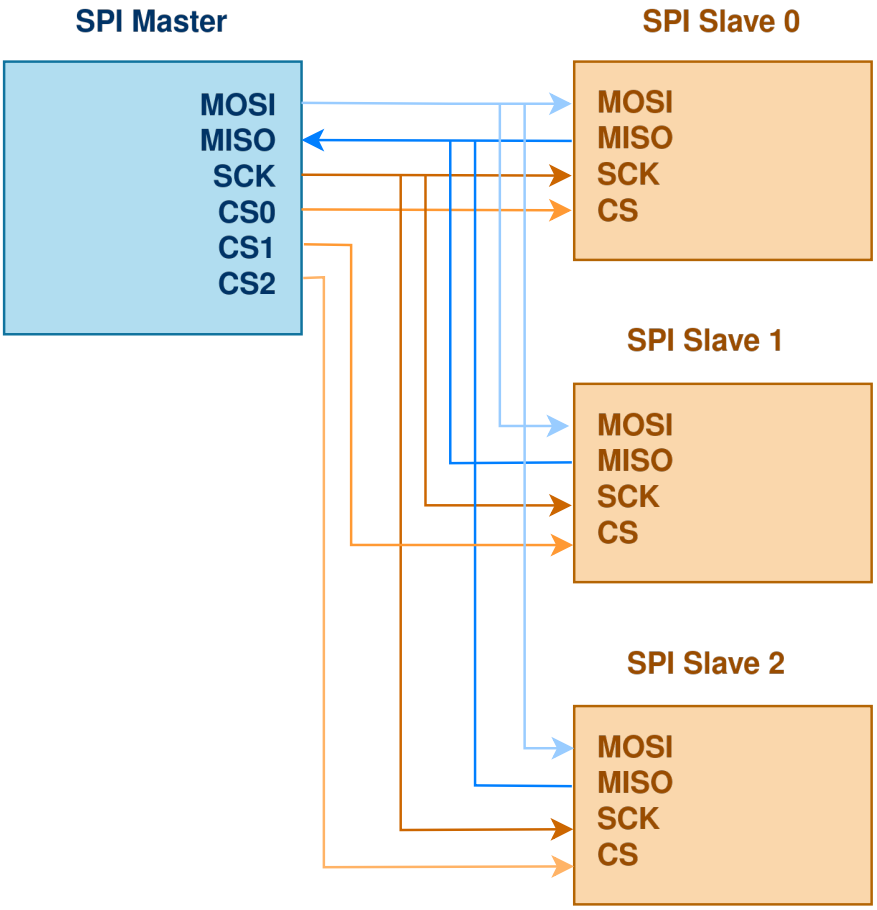


Figure 2.1: Example connection between a SPI master and three SPI slaves.

[1] full duplex-data can be transmitted and received simultaneously

The bus, as pictured in Figure 2.1, uses four signals for operation:

- **MOSI (Master Out, Slave In)** signal line used to transmit data from master to slave
- **MISO (Master In, Slave Out)** signal line used to transmit data from slave to master
- **SCK (Serial Clock Line)** signal generated by the master that is used to synchronize transfer between devices
- **CS (Chip Select)/SS (Slave Select)** signal used by the master to select which slave it is trying to communicate with

Every transfer is initiated by the master. To do so, it has to pull the CS line of the slave it wishes to communicate with low [2]. The serial clock edge synchronizes the shifting and sampling of the data. As SPI is full-duplex, both the master and slave can send and receive data at the same time.

Bus Configuration

For maximum flexibility, SPI offers different clocking configurations. The CPOL (Clock Polarity) parameter defines the idle level of the SCK line. CPOL 0 means the idle state of this line is 0 and the first rising edge after CS is asserted is the "leading edge„ that defines the first clock pulse. CPOL 1 means the idle state is 1 and the "leading edge" in this case is the first falling edge of the clock once CS is asserted.

The CPHA (Clock Phase) parameter defines which clock edge is the active edge that is used to read data. If CPHA is 0, data is captured on the leading edge of the clock signal, and for CPHA equal to 1, data is captured on the next edge after the leading edge. These combinations together result in 4 different communication modes offered by the bus.

- **Mode 0 (CPOL = 0, CPHA = 0)** the idle clock polarity is 0 and data is shifted on the rising edge of the clock signal.
- **Mode 1 (CPOL = 0, CPHA = 1)** the idle clock polarity is 0 and data is shifted on the falling edge of the clock signal.
- **Mode 2 (CPOL = 1, CPHA = 0)** the idle clock polarity is 1 and data is shifted on the rising edge of the clock signal.
- **Mode 3 (CPOL = 1, CPHA = 1)** the idle clock level is 1 and data is shifted on the falling edge of the clock signal.

A master/slave pair must use the same set of parameters: clock frequency, CPOL, and CPHA for data exchange to be possible.

SPI Frame

SPI does not impose a fixed structure for data frames unlike other protocols, such as I^2C. It does not use start or stop bits, meaning every bit transmitted is purely data. While SPI communication commonly employs 8-bit data units (bytes), it is flexible and can handle

[2] assuming the CS line is active low

arbitrary data sizes, including 16-bit, 32-bit, or even non-standard lengths, depending on the configuration of the master and slave devices.

As mentioned before, SPI also lacks built-in error detection or correction mechanisms, such as parity bits or cyclic redundancy checks. Consequently, any error-checking logic, such as checksums or retransmission protocols, must be implemented at the application layer if reliability is a concern.

2.1.2 I²C

I²C (Inter Integrated Circuit) is another serial synchronous communication protocol with similar use cases to SPI. It was developed by Phillips Semiconductors (now NXP Semiconductors) as a simple bidirectional 2-wire bus for efficient inter-IC control. It is a multi-master protocol, however, most applications employ only a single master in one network. If a device initiates transfers on the I²C bus, it is considered to be the bus master. Consequently, all other devices on the same network are referred to as slaves.

I²C operates in half-duplex mode, meaning data transmission occurs in only one direction at a time. Compared to SPI, I²C generally has lower data transfer speeds. The protocol specification defines several speed modes [7]:

- **Standard Mode (Sm)** allows speeds up to 100 kbit/s.
- **Fast Mode (Fm)** allows speeds up to 400 kbit/s.
- **Fast Mode Plus (Fm+)** allows speeds up to 1 Mbit/s.
- **High Speed Mode** allows speeds up to 3.4 Mbit/s.
- **Ultra Fast Mode** allows for speeds up to 5 Mbit/s, however the transfer is only unidirectional.

Physical Implementation

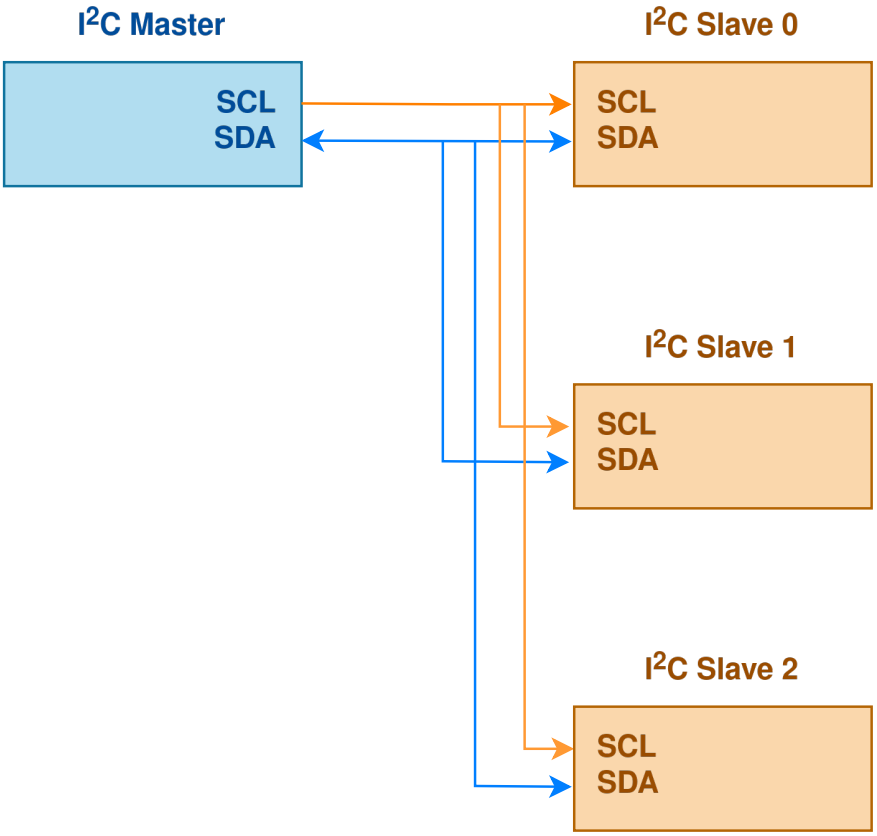


Figure 2.2: Example connection between I²C master and three I²C slaves.

As pictured in Figure 2.2, I²C uses two active signal wires that are both bidirectional.

- **SCL (Serial Clock Line)** the line that carries the clock signal generated by the master.
- **SDA (Serial Data Line)** the line used to transmit data for both master and slaves.

Both SCL and SDA lines are open-drain I/Os with pull-up resistors. This configuration allows multiple devices to share the same bus without electrical conflicts, as no device actively drives the lines high—only low signals are actively pulled to ground. The open-drain architecture ensures that a logic low (0) always takes precedence over a logic high (1), which is crucial for bus arbitration. Since I²C supports multiple masters on a single bus, this design naturally enables collision detection and arbitration, preventing data corruption when multiple devices attempt to communicate simultaneously. It does not use any Chip Select signal like SPI and theoretically speaking, it is possible to connect any number of devices together through these two lines.

I²C Frame

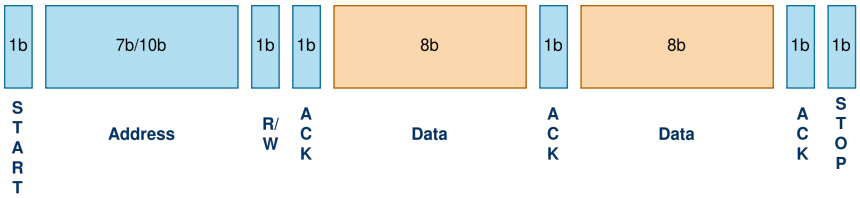


Figure 2.3: Composition of an I²C frame.

As pictured in Figure 2.3, the I²C standard specifies the following frame structure.

- **Start** The master initiates every transfer with a start condition, where SDA is pulled low and the start-bit is transferred while SCL remains high. I²C uses only a single start-bit with a value of zero.
- **Address** A 7 or 10 bit sequence that uniquely identifies each slave on the bus.
- **R/W** A single bit specifying whether the master is trying to read from or write data to the the selected slave.
- **ACK/NACK (Address)** The receiving device, whose address has been transmitted by the master confirms its presence on the bus by issuing a ACK bit-pulling the SDA line low. This tells the master that the slave it is trying to reach is available. If no slave devices recognizes the address, the result is a NACK. In this case, the master must abort the request as there is no one to communicate with and cannot generally be fixed by retrying.
- **Data** Data that are being transmitted, either from slave to master or from master to slave.
- **ACK/NACK (Data)** An acknowledgment bit is expected after every successfully transmitted data byte.
- **Stop** Every transfer is terminated with a stop condition, where master releases the SDA line back to high while the SCL line is high. I²C uses only a single stop-bit with a value of zero.

Addressing Modes

Conventionally, I²C uses a 7-bit addressing scheme, allowing up to 128 unique addresses on a single bus. However, this can be insufficient for applications that require a larger number of devices. To address this limitation, the I²C standard includes a 10-bit addressing mode, which expands the address space to 1,024 unique addresses. Devices with 7-bit and 10-bit addresses can be connected to the same I2C bus, and both 7-bit and 10-bit addressing can be used in all bus speed modes [7].

2.1.3 CAN

The CAN (Controller Area Network) bus is essential for modern automotive systems, enabling reliable and efficient communication between electronic control units. In modern

vehicles, CAN bus is what connects multiple nodes throughout the vehicle together. It was developed by BOSCH GmbH [12] with the purpose of reducing cost and complexity of automotive wiring as the number of computing devices in vehicles increased. Since then it has been adopted in various other areas, such as marine, medical, manufacturing, and aerospace and is now an ISO approved communication protocol (ISO 11898). This ISO standard describes how data is passed between devices on the network and defines the lowest two layers of the OSI model:

- **Physical Layer** Defining the transmission medium, signal levels and bit representation
- **Data Link Layer** Defining the message framing, arbitration, error detection and fault confinement, transfer rate and timing, message and status handling, etc.

The CAN bus is a serial, multi-master, message broadcast protocol. It functions as a peer-to-peer network where all devices within the network have equal rights to send messages over the bus. If multiple nodes attempt to transmit simultaneously, CAN uses a priority-based arbitration method to resolve conflicts. It offers signaling rates up to 1 Mbps, high immunity to electrical interference, and an ability to self-diagnose and repair data errors.

There are two versions of the CAN protocol differing in the size of the identifier: CAN 2.0A (Standard, most common in vehicles) with an 11 bit identifier and CAN 2.0B (Extended, usually used in trucks, buses and other large vehicles) with a 29 bit identifier. [2]

Physical implementation

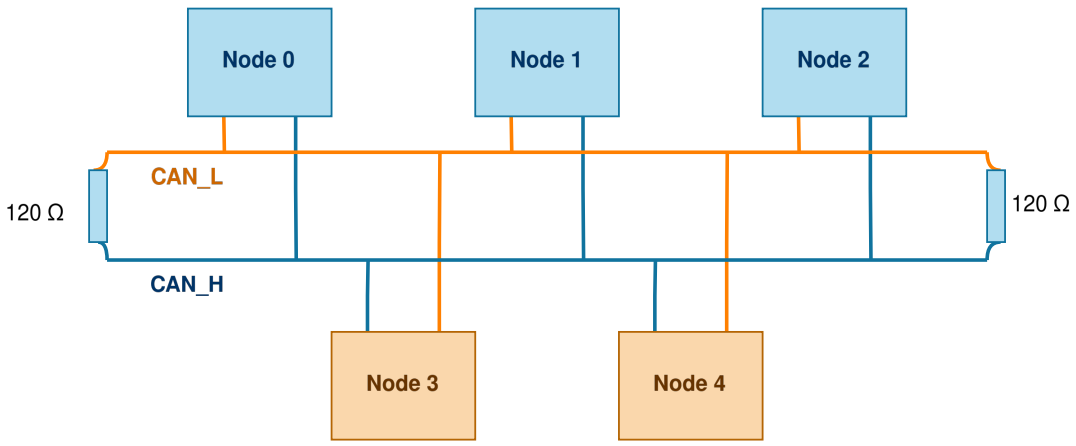


Figure 2.4: Example connection of 5 nodes in a CAN network.

The physical implementation of the bus, as pictured in Figure 2.4, compromises two wires, the CAN High (CAN_H, usually marked yellow in automotive) and the CAN Low (CAN_L, usually marked green in automotive) that are twisted together to prevent electromagnetic interference. A CAN network requires termination resistors of 120 Ohms between the CAN Low and High lines on each side of the bus. As these are connected in parallel, a correctly

configured CAN bus should have a resistance reading of around 60 Ohms between the Low and High lines.

CAN uses differential signaling, meaning that the current value on being sent over the network is obtained by subtracting the values of the High and Low lines. This helps in removing unwanted noise from disrupting the communication. A CAN line can be in two states-the recessive 1 or the dominant 0. The default state of the bus is recessive, meaning when the bus is inactive, we should always get a 1 reading. Another consequence of this setup is that during simultaneous transmission of dominant and recessive bits, the resulting bus value will be dominant. The termination resistors help keep this steady recessive level on the bus in rest state and also eliminate bus reflections.

CAN 2.0A Frame

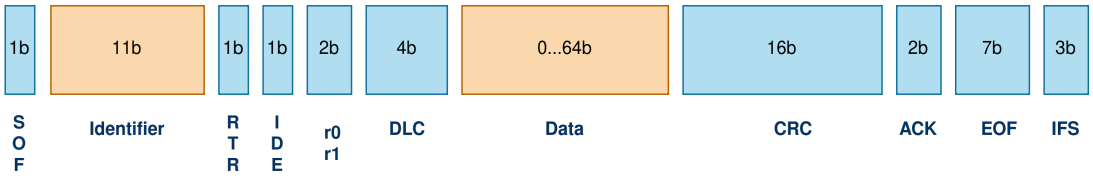


Figure 2.5: Standard CAN (CAN 2.0A) frame format.

The CAN 2.0A frame, as seen in Figure 2.5, consists of the following fields:

- **SOF (Start Of Frame)** Single dominant bit on the line marks the start of a message and is used to synchronize nodes on the bus.
- **Identifier** The 11-bit identifier identifies a node on the bus and determines its priority at the same time. The lower the identifier number, the higher the priority.
- **RTR (Remote Transmission Request)** Indicates whether a node sends data or requests dedicated data from another node.
- **IDE (Identifier Extension)** A single dominant bit which indicated that a standard CAN identifier with no extension is being transmitted.
- **r0, r1** Currently not used, reserved for future applications.
- **DLC (Data Length Code)** Determines the number of data bytes being sent.
- **Data** Up to 64 bits (8 bytes) of data can be transmitted in one message. Data is transferred MSB first.
- **CRC (Cyclic Redundancy Check)** Contains the 15 bit checksum of preceding data for error detection. The 16th bit of this field is the CRC delimeter, which has to be a recessive bit.
- **ACK (Acknowledge)** Indicates if the node has acknowledged and received the data correctly. This acknowledgment scheme solely enables the sender to know that at least one node on the bus, but not necessarily the intended recipient, confirmed the reception of the frame. The second bit of the 2 bit ACK field is the ACK delimeter, which is a single recessive bit.

- **EOF (End of Frame)** Each data frame and remote frame is delimited by a flag sequence consisting of seven recessive bits and marks the end of a CAN message.
- **IFS (Inter-Frame Space)** Consists of a three recessive bit. It is the time period between the end of one message and the start of the next message. The IFS is used to separate successive messages on the bus to prevent message collisions. During the intermission, no node is permitted to initiate a transmission. If a dominant bit is detected during the Intermission, an Overload Frame will be generated.

Frame types

The CAN standard specifies four different types of messages that can be sent over the bus, each carrying a different meaning.

- **Data Frame** is the most common type of CAN frame, it is used to transmit data over the bus.
- **Remote Frame** A remote frame is sent if a node requests data from another node. It has the RTR bit set as recessive and the DLC field contains the length of the expected response message.
- **Error Frame** is sent if the receiving or transmitting node detects an error and will cause all other nodes to also detect a fault. CAN has an elaborate system of error counters that ensures that a node can't flood the bus traffic by repeatedly transmitting error frames. An error frame consists of six consecutive bits of the same value, thus violating the bit stuffing rule.
- **Overload Frame** is sent by a node experiencing an overload, which means it is receiving frames faster than it is able to process them and requires extra time between successive data or remote frames.

Bus Arbitration

Any CAN node may start a transmission when the bus is idle. Possible conflicts, when multiple nodes start transmitting at the same time are resolved by a priority-based arbitration process, called the bit-by-bit arbitration. All nodes in the process of sending data also monitor the bus at the same time. If a node detects a dominant level when it is sending a recessive level itself, it will immediately quit the arbitration process and switch to just receiving instead. Since the identifier is transmitted MSB First [3], the node with the numerically lowest identifier field will gain access to the bus. A node that has lost the arbitration will wait until the bus becomes free before trying to transmit its frame again. The arbitration is performed over the whole Identifier Field and when that field has been sent, exactly one transmitter is left on the bus. This arbitration technique does not consume any bandwidth, the prioritized node continues the transmission as if nothing had happened.

Bit Stuffing

As there is no clock signal used in CAN, it implements a clever way for all nodes on the bus to get the clock from the waveform. Therefore, receivers use the transition in the

[3] Most Significant Bit First

waveform to synchronize the receiver clocks with the transmitter. To ensure there are enough transition for this to be possible, CAN implements a mechanism called bit stuffing. Bit stuffing inserts an extra bit of the opposite logic level after five consecutive bits of the same logic level were transmitted. These extra stuff bits are automatically discarded by the receiver so the application software in the CAN node never sees them.

CAN FD

CAN FD (CAN Flexible Data-rate)[4] is an extension of the original CAN bus protocol, designed to increase data transfer rates and message sizes for use in modern automotive systems. CAN FD allows for data transmission at speeds of up to 5 Mbit/s and supports a payload of up to 64 bytes, as opposed to the 8-byte limit of traditional CAN. It is typically used in high performance ECUs of modern vehicles. Another pleasant feature of CAN FD is that it is backwards compatible with Standard CAN, and therefore CAN FD ECUs and Standard CAN ECUs can be mixed together without much overhead.

Error Handling

CAN is considered a very reliable and robust communication protocol thanks to its abundant error checking mechanisms. Altogether, CAN incorporates 5 different techniques to ensure data is received correctly. If a message fails with any one of these error detection methods, it is not accepted and an error frame is generated from the receiving nodes, causing the transmitting node to resend the message until it is received correctly [13].

At the message level, we have the CRC field, the Acknowledgment field, and the Frame Check. A CRC error is raised by a receiving node whenever the calculated CRC differs from the actual CRC in the frame. A acknowledgment error on the other hand is raised by the transmiting device in the event that it does not detect a dominant Acknowledgment Bit (ACK), meaning no device on the bus acknowledged reception of the frame. The frame check looks for fields in the message, such as SOF, EOF or ACK delimiter, which must always be recessive bits and if a dominant bit is detected, a frame error is generated.

At the bit level, every bit sent on the bus is monitored by the transmitter and compared with the intended value. In case of a mismatch, it raises an error. The only exception to this error detection mechanism is when the identifier field, which is used for bus arbitration, or the acknowledgment field, which requires a recessive bit to be overwritten by a dominant bit, are transmitted.

CAN also ensures compliance with the bit-stuffing rule and if it is violated, meaning the required bit is not transmitted, an error is detected and reported.

CAN nodes are able to distinguish short disturbances from permanent failures by implementing a smart error counter mechanism for each node. Defective nodes are switched off.

2.1.4 LIN

Not all nodes require a large bandwidth, such as the one offered by CAN. This lead to the design of low-cost alternatives for components with considerably low speed and low bandwidth requirements that are mainly used in areas where performance and speed is not a critical issue. This applies for tasks such as the circuits controlling windscreen wipers, air

[4] https://www.csselectronics.com/pages/can-fd-flexible-data-rate-intro

conditioning or windows/mirrors. One such alternative is LIN (Local Interconnect Network) [15].

LIN protocol is another automotive applications protocol used for low-speed, cost-effective communication. It was developed by the LIN consortium, consisting of several major automakers, including Mercedes, BMW, Volkswagen, Audi and Volvo. It is a broadcasting, single-ended, serial one-wire interface typically implemented as a sub-bus of CAN and offers communication at baud rates of up to 20 kbps. It follows a single-master, multiple-slave architecture, where a master node is responsible for initiating all communication, while slave devices respond accordingly. [3]

Physical Implementation

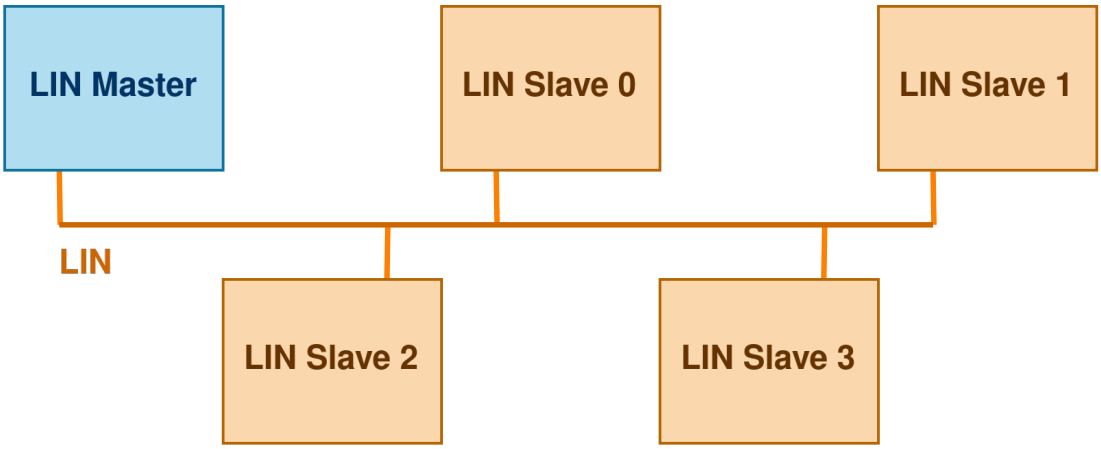


Figure 2.6: Example connection of 5 nodes in a LIN bus.

The physical implementation of the LIN protocol, as pictured in Figure 2.6, requires only a single-wire for its serial communication. The LIN bus operates between 9 to 18 volts, although the standard voltage for LIN bus systems is 12 volts [8], as it is the norm for most conventional automotive electrical systems. All devices on the bus must share a ground reference. The LIN bus does not require termination resistors like CAN. The master and slave devices are all connected to the same wire via pull-up resistors to maintain signal integrity and ensure the bus remains in its default state when no device is transmitting. The values are typically 30kΩ for slave devices and 1kΩ for the master device.

The signal level on the shared bus line can be either recessive or dominant. A recessive value is logic 1; the default state of the bus, where the bus is pulled to V_BAT through the pullup resistor. The dominant state is logic 0, which is achieved when a device actively pulls the bus to ground (0V) to transmit data.

LIN Frame

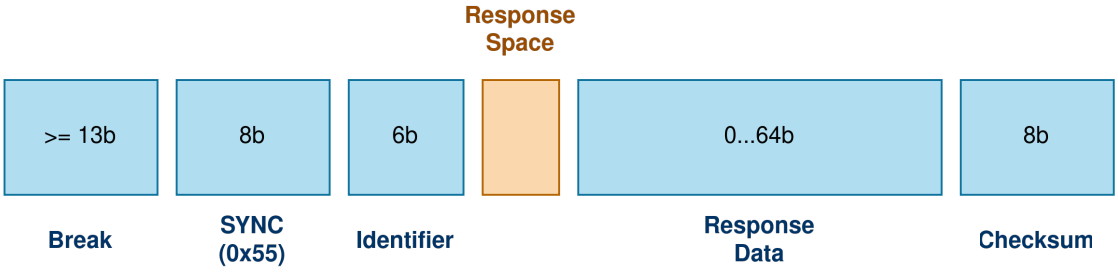


Figure 2.7: Standard LIN frame format

The LIN frame, as pictured in Figure 2.7, contains the following fields:

- **Break** The break field is used to signal the beginning of a frame. It contains at least 13 dominant bits (logical value zero) followed by one recessive bit (logical value one) as a break delimiter. It is always generated by the master.
- **SYNC** The sync byte is a field with value of 0x55 and allows the slave nodes to detect the beginning of a new frame and be synchronized at the start of the identifier field.
- **Protected Identifier (PID)** The protected identifier is composed of two subfields: the first 6 bits are used to encode the frame identifier and the last 2 bits the identifier parity. The frame identifier uniquely defines the purpose of the frame
- **Response Space** A short idle period after the PID before the slave starts transmitting data. It allows the addressed slave time to prepare and start transmission.
- **Response Data** A frame carries between one and eight bytes of data. Data is sent LSB first.
- **Checksum** The checksum allows the receiver to detect any bits that may have been inverted during transmission. LIN provides two methods for checksum calculation; the classic checksum, which is computed only over the data field, and the enhanced checksum, which also includes the protected identifier in the calculation.

All messages are initiated by the master, with at most one slave replying to a given message identifier. The master node can also function as a slave by replying to its messages. As the master initiates all communications, it is not necessary to implement collision detection.

Error Handling

Although the LIN protocol is designed primarily for non-critical automotive applications unlike CAN, it still incorporates several safety measures to enhance reliability and prevent communication errors. As mentioned before, the checksum field of each frame verifies data integrity. The parity check of the Protected Identifier helps detect single-bit errors in the identifier field. Also, if a slave fails to respond within a predefined time, the master may retransmit the request or trigger an error handling routine.

2.2 Safety Measures using Watchdogs

As mentioned earlier, most automotive communication protocols designed with safety in mind already include robust error detection and fault confinement mechanisms. However, additional safety measures can be put in place to further enhance reliability.

A watchdog is a fail-safe monitoring system that continuously checks the functionality of electronic control units and software. If a system becomes unresponsive or enters an undefined state, the watchdog triggers corrective actions to restore normal operation, preventing potential failures from escalating.

There are several types of watchdog implementations, but the most notable categories are [5]:

- **Hardware Watchdog** A hardware watchdog runs independently of the ECU and the main program. It is usually realized by a special hardware chip or module with independent power supply and clock source. The main program keeps periodically „feeding the watchdog“ (by sending a specified signal) to keep the watchdog active. If the system doesn't repeat this process within a set time frame, the watchdog assumes something is wrong and forces a reset.
- **Software Watchdog** A software watchdog is implemented purely within the firmware and runs as part of the operating system or application code. It monitors critical tasks and can trigger a system reset if they fail to respond.

By integrating watchdogs into automotive communication systems, vehicles can detect and recover from failures before they compromise safety or performance.

LIMP Home Mode

Some devices enter the LIMP home mode when they detect a fault such as a failed watchdog refresh. It is a fail-safe state, which allows the system to continue operating in a limited capacity to ensure safety and prevent damage.

Watchdogs in Automotive Systems

In automotive systems, watchdogs are commonly used to monitor the operation of safety-critical control units. For instance, a watchdog may be integrated into an ECU responsible for managing essential functions such as braking, steering assistance, or communication between subsystems.

During regular operation, the ECU periodically resets the watchdog timer to indicate that it is functioning correctly. If a fault occurs—such as the software becoming unresponsive, entering an unexpected loop, or encountering a deadlock—the watchdog will not be refreshed within the expected timeframe. As a result, the watchdog triggers a system reset or initiates a predefined recovery mechanism, such as entering the LIMP home mode. This process ensures that the control unit returns to a known and stable state, minimizing the risk of unsafe behavior.

2.3 Automotive Software Development Tools

This section provides a brief overview of tools used in automotive software development.

[5] https://www.pusr.com/blog/Difference-between-software-watchdog-and-hardware-watchdog

2.3.1 AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) is a standardization initiative of leading automotive manufacturers and suppliers that was founded in 2003. The goal is the development of a reference architecture for ECU software that can manage the growing complexity of ECUs in modern vehicles. The AUTOSAR standard is designed to enable software standardization, modularity, reusability, and flexibility.

AUTOSAR Layered Architecture

AUTOSAR follows a structured architecture comprising several layers, as in Figure 2.8, each serving a different purpose:

- **Application Layer** The topmost layer where all application software components (SWCs) reside. It is hardware-independent and focuses solely on the functionality and output of the applications. The SWCs do not need to know about the underlying hardware or communication protocols.
- **Runtime Environment (RTE)** The RTE serves as an intermediary, enabling communication between the application layer and the basic software layer. It abstracts the details of the underlying hardware and allows SWCs to interact with each other and with services provided by the basic software.
- **Basic Software (BSW)** This is the foundational layer that provides essential services to the application layer. It includes various sub-layers such as the MCAL (Microcontroller Abstraction Layer), which interfaces directly with microcontroller peripherals, ensuring that higher layers are independent of specific microcontroller implementations. The ECU Abstraction Layer provides an interface that abstracts the ECU hardware specifics from upper layers. The Services Layer offers common services like communication, diagnostics, and memory management to support application needs. Finally, the Complex Drivers Layer handles specialized functions that require direct access to hardware components not covered by other layers.

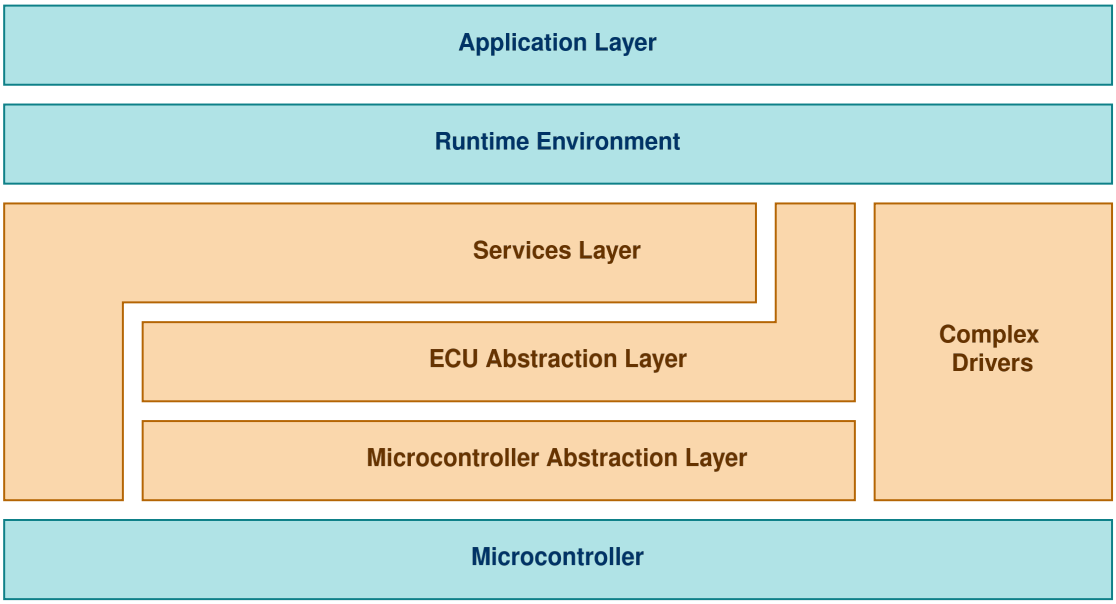


Figure 2.8: AUTOSAR layered architecture

As of 2017, AUTOSAR is being developed in two variants:

- **Classic Platform:** Focused on traditional vehicle systems, suitable for real-time, deterministic and safety-critical applications. It is typically used for power train and chassis functionalities like engine control, transmission control or braking systems [13].
- **Adaptive Platform:** Designed for more complex applications, including those requiring high computational power and flexibility. It is typically used in advanced driver assistance systems (ADAS), autonomous driving or infotainment [13].

Both variants continue to use the layered architecture, but for the Adaptive Platform, it is adapted and evolved to meet the needs of more complex and dynamic systems.

2.3.2 NXP Real Time Drivers

NXP's Real-Time Drivers (RTD) are production-qualified software abstractions designed for complex hardware features, suitable for use in both AUTOSAR and non-AUTOSAR applications1. RTD provides standardized APIs that are consistent across different products, with dedicated hardware-specific interfaces. These drivers offer multiple software features that extend the AUTOSAR standard, fully covering hardware features and peripherals1. RTD combines elements of NXP's SDK and AUTOSAR MCAL drivers with new complex device drivers into a single software product

2.3.3 S32 Design Studio

NXP's S32 Design Studio is an integrated development environment (IDE) for automotive and ultra-reliable Arm®-based microcontrollers and processors. The platform offers advanced tools for software development, including debugging, code generation, and device

configuration, providing a seamless workflow for designing and optimizing embedded applications. It supports both real-time and safety-critical environments, with features like support for AUTOSAR and compliance with functional safety standards (ISO 26262). It is based on open-source software including Eclipse IDE, GNU Compiler Collection (GCC) and GNU Debugger (GDB) and is integrated with NXP's software and hardware development tools.

S32 DS Configuration Tools

The S32 Design Studio provides a suite of configuration tools to assist users in setting up devices and drivers, including the Peripherals Tool, Clock Tool, and Pins Tool. These tools offer a graphical user interface (GUI) that simplifies the configuration of the MCU and other project components. The configuration settings are stored in an XML file, which can then be used to generate driver code for the final application.

Chapter 3

Concept and Design of the Demonstration Platform

This chapter presents the concept of the demonstration platform developed for this thesis. It outlines the key requirements, the intended functionalities and the overall system design.

3.1 Requirements

Before we dive into the solution, it's essential to begin with a few requirements outlining the desired characteristics of the future implementation. These will serve as the foundation for our upcoming work. These can be categorized as functional requirements and hardware requirements.

3.1.1 Functional Requirements

These requirements should ensure that the platform effectively teaches users about automotive communication protocols, facilitates hands-on experimentation, and provides a user-friendly learning experience.

- **Protocol Support** The platform should include multiple microcontrollers communicating via automotive communication protocols that we want to demonstrate. CAN should be chosen as the primary backbone for ECU to ECU communication, meanwhile LIN will handle some low-speed subsystem control. SPI and I^2C will be used for MCU to peripheral communication.
- **User-friendly Interface and Interactivity** Users should be able to interact with the platform and also get feedback from their actions. These interactions should be intuitive and easy to understand.
- **Simulation and Visualization Tools** Interactive controls will trigger different realistic automotive scenarios, such as simulating the lighting system or the parking assistant.
- **Aesthetic, Compact and Transportable** As this demonstration platform is mainly intended to be used at different events that the company will be participating in, it is important for the final product to be easy to move from place to place and to set up. Visually appealing design should also be taken into account, as the platform also serves a representative purpose.

3.1.2 Hardware Requirements

The platform's hardware is designed to simulate different automotive scenarios while allowing the user to interact with it. To achieve this, the key hardware components should include:

- **Core Processing Units** The system needs a powerful enough processing unit to run all selected automotive scenarios in real-time. It should support the necessary automotive communication interfaces and have enough computing power to handle all tasks smoothly.
- **Human-Machine Interface (HMI)** A control panel will serve as the main point of interaction between the user and the platform. It should allow users to trigger different scenarios, view outputs, and monitor system behavior easily.
- **Sensors and Peripherals** Various sensors and additional components will help mimic real vehicle functions, making the simulation more realistic and interactive.
- **Actuators and Output Components** To bring the simulations to life, the platform will use output devices like actuators and indicators. For example, lights could turn on or off to demonstrate how a real vehicle would respond in a given scenario.

3.2 Proposed Solution

For simulation, I chose the following scenarios.

- **Lighting system** The platform will include a system simulating lights in a traditional car, including headlights, brake lights and turn indicator lights.
- **Parking assistant** The platform will visualize input from proximity sensors to simulate a simple parking assistant setup.
- **Trunk opening mechanism** The platform will include a simulation of automatized trunk opening via a pushbutton.
- **System reaction to a fault** The platform will allow the user to simulate a fault in the system and visualize its reaction to it.

The architecture will consist of two main ECUs, each representing a different automotive cluster. These ECUs will communicate via CAN, which serves as the backbone protocol for automotive communication. Additionally, two lower-performance ECUs will function as smart sensors, collecting data from proximity sensors connected via I^2C. These sensors will preprocess and aggregate the data before transmitting it to the main cluster ECU over LIN. The main ECUs will have distinct roles: one will manage output devices and actuators, while the other will handle user input and the power system basis chip (SBC).

All devices in the system will communicate using different protocols, but a key objective of this project is to make these interactions visually comprehensible. To achieve this, LED paths will be used to represent the flow of data between devices, providing an intuitive visualization of the communication process. Each protocol will be assigned a distinct LED color, allowing users to easily differentiate between them.

3.2.1 Hardware Design

In order to meet the hardware requirements mentioned previously, the following components were chosen for the implementation.

Processing Units

- **S32K312EVB-Q172**

  The S32K312EVB-Q172 is an evaluation and development board designed for general-purpose industrial and automotive applications. It is powered by the 32-bit Arm® Cortex®-M7 S32K3 MCU, providing high-performance processing capabilities [11]. The board comes equipped with a TJA1043 CAN transceiver and a TJA1022 dual LIN transceiver, enabling robust communication across automotive networks. Additionally, it features the FS26 System Basis Chip for efficient power management, ensuring reliable operation in embedded applications. In this project, we will use two of these to serve as the pair of main ECUs.

- **S32K144EVB-Q100**

  The S32K144EVB is a cost-effective evaluation and development board also designed for general-purpose industrial and automotive applications. Built around the 32-bit Arm® Cortex®-M4F S32K14 MCU, it offers reliable performance for embedded systems [5]. The board includes a TJA1027 LIN transceiver, enabling seamless communication within a LIN network. In this project, two S32K144EVB boards will function as smart sensors, gathering data from proximity sensors, processing the information, and transmitting it to the S32K312EVB-Q172 over LIN for further evaluation.

Peripherals

- **FRDM-XS2410EVB Evaluation Board**

  The FRDM-XS2410EVB Freedom board is a low cost and easy to use evaluation board featuring the MC33XS2410 chip, a four-channel self-protected high-side switch designed for robust performance in various applications. Each of its four outputs is fully protected and configurable, capable of delivering continuous currents of up to 2 A per output. The device operates over a wide voltage range, from 3.0 V to 60 V, which makes it versatile across different power systems. Configuration and control is managed through SPI. Whenever communication with the MCU is lost, the device enters a safe operation mode, but remains operational, controllable and protected. In this project, it will serve as the controller for the lighting system.

- **FRDM-HB2001-EVM Evaluation Board**

  The FRDM-HB2001-EVM is a user-friendly evaluation kit featuring the MC33HB2001 chip, a 10 A H-Bridge, SPI-programmable brushed DC motor driver. This board is primarily designed for DC or servo motor control applications. In this project, it will drive a small DC motor responsible for the trunk opening mechanism. The MC33HB2001 offers flexible control options, allowing operation through direct input pins or via SPI, which will be used in this project. It will be in charge of driving the motor in the trunk opening mechanism.

- **FRDM-CD1020-EVM Evaluation Board**

The FRDM-CD1020-EVM is an evaluation board built around the CD1020 chip, a 22-channel switch detection interface designed to monitor the opening and closing of up to 22 switches. It communicates with the controlling MCU via SPI, ensuring efficient data transfer. Operating within a 6.0V to 36.0V range, it is well-suited for automotive and industrial applications. Additionally, it features an active interrupt pin (INT_B), which triggers whenever a change in the switch state occurs, enabling real-time event detection. In this project, it will be responsible for detecting user-pressed switches.

- **HC-SR04**

  The HC-SR04 is low-cost ultrasonic distance sensor commonly used for measuring distances with high accuracy. It operates by emitting an ultrasonic pulse and measuring the time it takes for the echo to return after bouncing off an object. The sensor consists of a transmitter and a receiver, allowing it to detect objects within a range of 2 cm up to 400 cm. While it supports multiple communication modes, this project utilizes the I²C mode, enabling seamless integration with the S32K144EVB-Q100 for obstacle detection.

- **DOGXL160-7 Display**

  The DOGXL160-7 is a 160x107 pixel graphic display driven by the UC1610 controller, offering flexible communication via SPI or I²C. In this project, a module featuring two of these displays will be used to visualize data collected from the parking sensors—one dedicated to the rear and the other to the front, providing clear and real-time feedback on obstacle detection.

- **APA102 Addressable LEDs**

  To visualize the flow of communication, I will use APA102 LED strips. These are individually addressable RGB LEDs that operate via a two-wire interface similar to SPI, consisting of a clock signal and a data signal for precise control.

3.3 Physical Model

A key aspect of this project is its representative design. As previously mentioned, the final product must be compact and transportable, so it has been designed with these requirements in mind. To emphasize that we are dealing with automotive environment, the top cover will have the form of a car model.

The control panel serves as the user interface. As shown in Figure 3.1, a significant portion of the panel is occupied by two displays—one displaying proximity data from the front sensors and the other from the rear sensors. On the right side of the displays, a section is dedicated to push buttons that trigger various simulations, such as activating hazard lights, opening the trunk, and turning on the headlights. Above these buttons, a three-state switch allows the user to activate the turning indicators for both directions. On the opposite side of the panel, two additional push buttons are present—one for triggering the SBC Watchdog failure and another for triggering the HSS Watchdog failure.

Another major focus of the design is the platform beneath the car model, where communication visualization takes place. LED paths will illuminate dynamically to represent data flow between ECUs, with distinct colors indicating different communication protocols. Additionally, the trunk opening mechanism allows the user to lift the car model, providing a better view of the platform and its visualized communication paths.

To maintain a clean and compact appearance, all electronic components will be concealed within the structure.

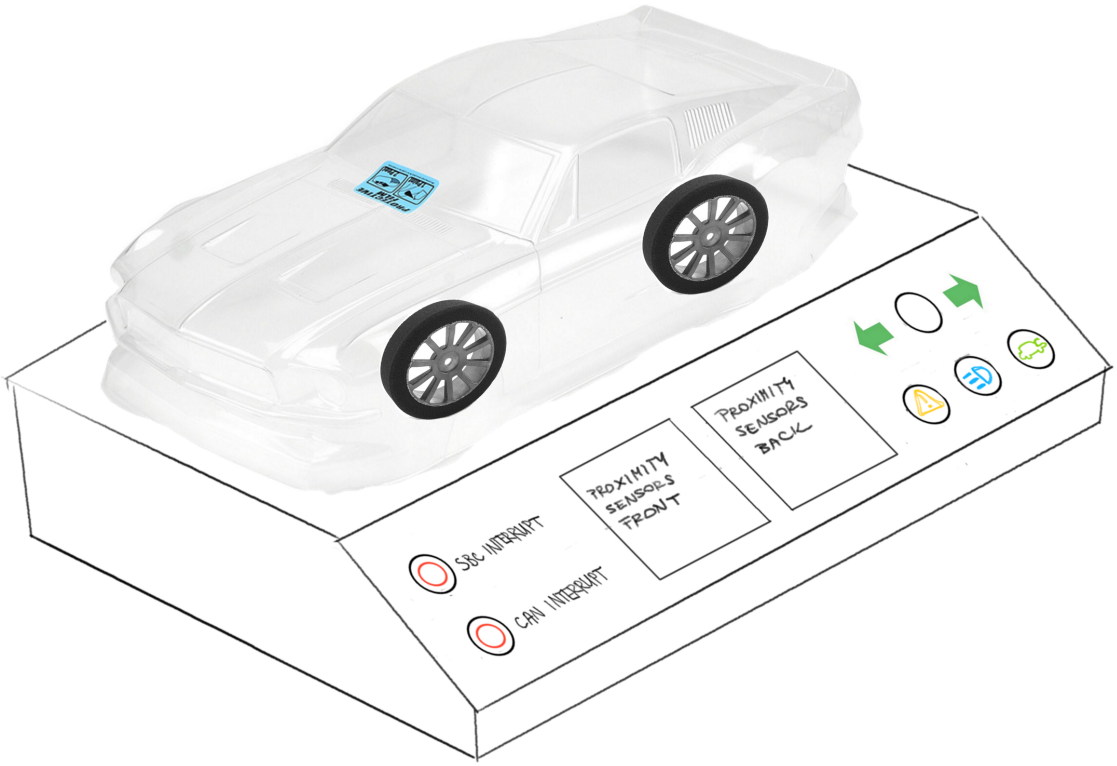


Figure 3.1: Conceptual design of the physical model.

Chapter 4

Implementation

This chapter describes the individual stages of the implementation process, beginning with the hardware design, followed by the software development, and concluding with the construction of the physical demonstration platform.

4.1 Hardware components and assembly

The hardware design of this project can be categorized into three distinct groups, each centered around a microcontroller responsible for executing the corresponding functionality.

4.1.1 S32K144 Proximity Data Collector Circuit

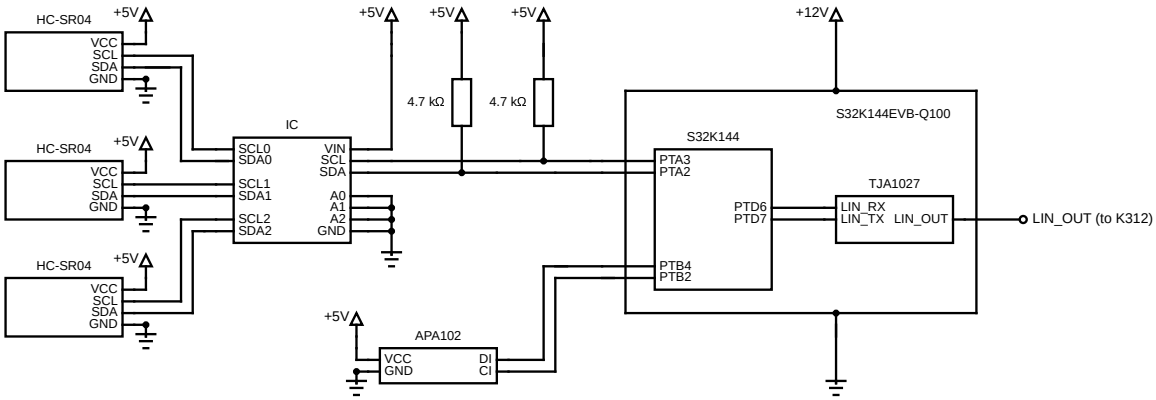


Figure 4.1: Hardware connections of the S32K144 cluster.

The two S32K144EVB-Q100 development boards are each responsible for communicating with three proximity sensors (Figure 4.1). However, these sensors have a fixed I²C address, which prevents them from operating on the same I²C bus simultaneously. To overcome this limitation, an I²C multiplexer (TCA9548A) is integrated into the circuit.

The TCA9548A multiplexer features a configurable I²C address, which can be set using the A0, A1, and A2 pins. In this application, all three pins are connected to ground, assigning the device an I²C address of 0x70, as specified in [4]. Each of the three HC-SR04 sensors is connected to one of the first three I²C channels of the multiplexer via the respective SDAx and SCLx lines. Both the sensors and the multiplexer require an external

5V power supply, meanwhile the S32K144EVB-Q100 is powered with 12V through the barrel connector.

To ensure proper communication, the SCL and SDA lines of the multiplexer are connected to the PTA3 and PTA2 pins of the S32K144EVB-Q100 through 4.7k Ohm pull-up resistors. The S32K144EVB-Q100 board features an onboard TJA1027 LIN transceiver, which is routed to the corresponding LIN_RX/TX pins. To enable LIN communication, these pins must be configured to use the LPUART2 interface. The LIN_OUT output of the LIN transceiver is then connected to the LIN interface of the S32K312EVB-Q172, enabling communication between the two microcontrollers.

4.1.2 S32K312 Output Peripheral Circuit

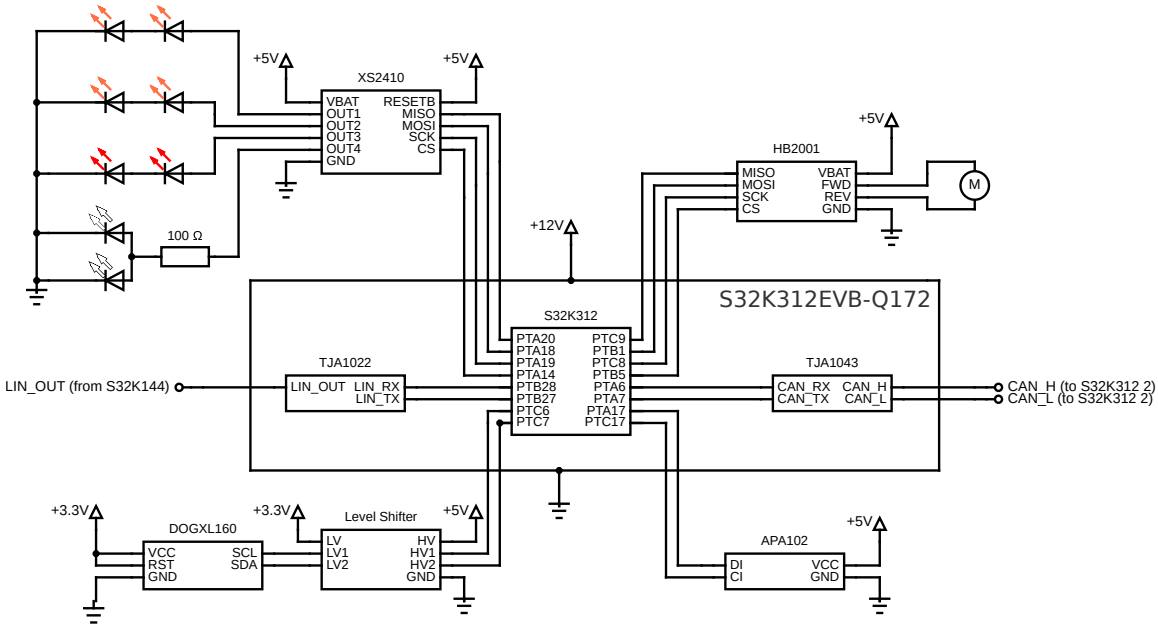


Figure 4.2: Hardware connections of the first S32K312 cluster.

Figure 4.2 illustrates the wiring of the first of two S32K312EVB-Q172 boards, which is responsible for controlling the output devices.

One of these output devices is the HB2001 motor controller, which is connected to the board's LPSPI0 peripheral. The motor itself is connected to the REV and FWD pins, which determine the motor's direction of rotation.

Another key component is the XS2410 High-Side Switch (HSS), which controls all the LEDs in the car's lighting system. This switch is connected to the LPSPI1 peripheral. Its four outputs serve different functions:

- **Output 1 & Output 2** control the left and right turn indicators. Each contains two orange LEDs connected in series.
- **Output 3** controls the brake lights. These consist of two red LEDs in series.
- **Output 4** controls the headlights, which consist of two white LEDs. Unlike the red and orange LEDs, the white LEDs require a higher voltage drop (3.1V instead

of 2.3V). To ensure they shine brightly with the 5V supply, they are connected in parallel rather than series.

Additionally, the board displays proximity sensor data collected from the front S32K144 board on a DOGXL160 display. Since the display operates at 3.3V logic while the board uses 5V logic, a level shifter is used for compatibility.

For communication interfaces, the TJA1022 LIN transceiver communicates via the LIN_RX/TX pins, which are internally linked on the board. However, the correct peripheral must be mapped to these pins—in this case, LPUART5. The same applies to the TJA1043 CAN transceiver, which requires proper routing for the CAN_RX/TX pins to the FLEXCAN0 peripheral.

Finally, the APA102 LED strip is controlled via the LPSPI3 peripheral. Unlike traditional SPI, it only requires a clock and data signal to function. This board manages a 17-LED strip to visually represent communication between the key components.

4.1.3 S32K312 Input Peripheral Circuit

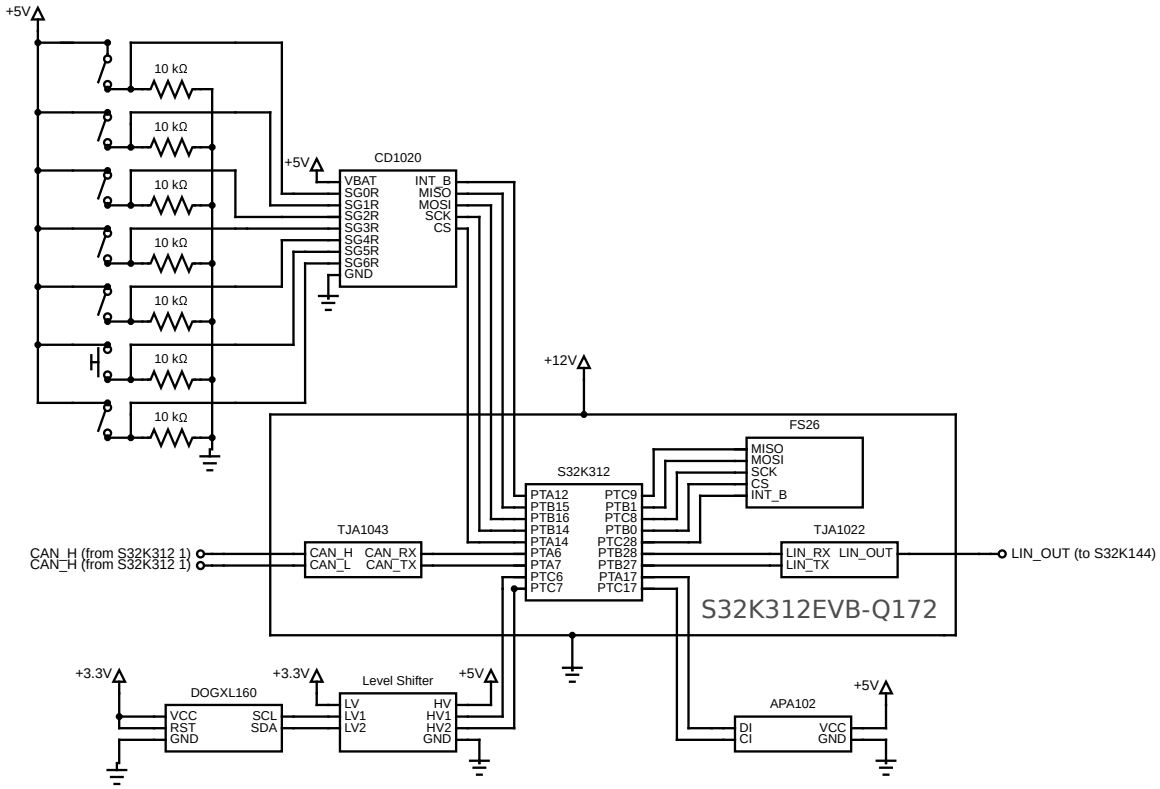


Figure 4.3: Hardware connections of the second S32K312 cluster.

Figure 4.3 illustrates the circuit connections for the second S32K312EVB-Q172 board, which is responsible for the input devices. Many peripherals remain unchanged from the first board, including the display, LED strip, CAN and LIN transceivers. However, this board also includes the CD1020 MSDI [1], which monitors seven user input switches—six latching buttons and one momentary push button. In addition to the SPI interface routed to the

[1] Multiple Switch Detection Interface

LPSPI1 peripheral, the CD1020 is connected via the INT_B pin, which notifies the board whenever a switch state changes.

Another key difference is the inclusion of the FS26 power management board, which is internally routed to the LPSPI0 peripheral. This component is responsible for supplying power to the board and ensuring correct voltage levels. It also features an INT_B pin, which is pulled low in the event of an error—such as a watchdog failure leading to a loss of communication.

4.2 Software Development

The solution is structured into three separate source code projects, namely *S32K144_Demo*, *S32K312_Demo_1*, and *S32K312_Demo_2*. All code is written in C, as the main language for embedded development. Each project includes the configuration and source code specific to its respective device. The two S32K144 boards share the same code, while each S32K312 board runs its own distinct implementation. Development was carried out using S32 Design Studio 3.5 and device configuration was performed using the S32DS Config Tool, where all necessary AUTOSAR components were selected for each project.

The contents of all the projects follow the same structure. The *generate/include* and *generate/src* folders contain all the configuration files produced by the Config Tool, including headers and source code. The *RTD/include* and *RTD/src* store AUTOSAR component drivers imported through the Config Tool. The *board/* folder contains specific configurations for the distinct MCUs. The *src/* folder contains the source code of the implementation, the *main.c* within this folder includes the core implementation that will be executed by the MCU.

4.2.1 S32K144 Proximity Data Collector Firmware

This section briefly explains the principles of operation behind the firmware for the S32K144 board. The implementation for this section is contained within the *S32K144_Demo* project.

Initialization

The first section of *main.c* contains initialization of all required AUTOSAR drivers and peripherals. The essential drivers include Mcu, Port, and Platform, which handle core board functions such as pin multiplexing, clock configuration, and interrupt management. Next is the GPT (General Purpose Timer) driver, which configures the user defined timers and the SPI driver, I^2C and LIN, which are required for the communication interfaces.

Main Loop

The initialization sequence is followed by an endless while loop where the three proximity sensors are continuously polled for new data. Each reading request consists of three steps, which are implemented in the *distanceFromSensor* function:

1. Selecting the correct sensor by sending an I^2C message to the I^2C multiplexer.
2. Sending a measurement request signal to the sensor to trigger a new reading.
3. After a short delay, polling the sensor again to retrieve the measurement result, which consists of three bytes.

Algorithm 1 The main loop of the S32K144 board firmware.

```
1: zones[0..3] = ZONE_NONE
2: while true do
3:     data_changed ← false
4:     for sensor ← 0..3 do
5:         distance ← distanceFromSensor(sensor)
6:         zones[sensor] ← convertToZone(distance)
7:         data_changed ← zone[sensor]_k ≠ zone[sensor]_{k−1}
8:
9:     end for
10:    if data_changed == true then
11:        sendLINFrame(zones[0..3])
12:    end if
13: end while
```

The raw sensor data is then converted into a human-readable unit (centimeters) using formula 4.1.

$$result\_in\_cm = \frac{(data[0] << 16) + (data[1] << 8) + data[2]}{10000} \tag{4.1}$$

The resulting value is then sent back to the main application.

In order to minimize the amount of data that needs to be transmitted, the firmware organizes the sensor readings into groups called zones. It pre-processes the data so that it can be directly utilized by the visualization component. The function convertToZone handles this conversion. The application distinguishes four distinct zones:

- ***ZONE_A*** Measurement reading of less than 5 centimeters.
- ***ZONE_B*** Measurement reading of less than 10 centimeters.
- ***ZONE_C*** Measurement reading of less than 15 centimeters.
- ***ZONE_NONE*** Any reading greater than 15 centimeters.

All functions regarding the HC-SR04 sensor interface are implemented in the *Proximity_Sensors.c/.h* files.

After fresh measurements from all sensors are stored, the device checks if any values have changed since the last iteration. If a change is detected, a LIN frame is assembled and sent over the LIN bus to the S32K312 board.

LED Control Timer

Another key task of this device is to visualize I^2C communication between the board and the three proximity sensors. To achieve this, three LED paths are used, each leading from a sensor to the block representing the device.

This functionality is implemented using a timer that triggers an interrupt every second. The callback of this interrupt is the *RGBLed_Refresh* function, which updates the state of the LEDs every time it is invoked. Since data transmission is continuous, the LED paths remain constantly illuminated, providing a real-time visual representation of the ongoing communication. All functions related to LED control are in the files *RGB_Leds.c/.h*

4.2.2 S32K312 Output Peripheral Firmware

This section describes the firmware functionality for the first S32K312 board. All related files are contained within the *S32K312_Demo_1* project.

Initialization

The execution of the driver code begins once again with an initialization phase for all the required AUTOSAR components. These include the Mcu, Port, and Platform modules, along with the communication interfaces: SPI, I^2C, CAN, and LIN. Additionally, the Gpt module is used for user-defined timers, while Icu handles user-defined interrupt inputs. The initialization phase also involves configuring the necessary registers of the XS2410 and HB2001 devices, and turning off all LEDs to ensure a defined starting state.

Main Loop

The firmware once again exhibits an endless loop, which continuosly handles all functionality.

Algorithm 2 The main loop of the first S32K312 board firmware.

```
1: display_refresh ← false
2: Can_rxFlag ← false
3: while true do
4:     if display_refresh == true then
5:         Display_Refresh(LIN_DATA)
6:         display_refresh ← false
7:     end if
8:     if Can_rxFlag == true then
9:         Can_updateOnReceivedData(CAN_DATA)
10:        Can_rxFlag ← false
11:    end if
12:    XS2410_updateOnReceivedData()
13: end while
```

The loop continuously checks if any flags are set and reacts accordingly. Flags are set within the interrupt handler routines of the CAN and LIN transceivers. The display refresh process consists of two phases: first, the buffer is populated with bitmap data indicating which zones should light up, and then this buffer is transmitted over I^2C to the display.

The function *Can_updateOnReceivedData* iterates through the received CAN buffer and, based on the data, sets the corresponding flags that are later used by the function *XS2410_updateOnReceivedData*. Additionally, it initiates the motor motion for the trunk opening functionality if it is requested and feasible to execute.

CAN Callback Function

The *CanIf_RxIndication* function is called whenever new CAN data is received from the other S32K312, containing updated information. The received data is copied from the receive buffer into the corresponding variable (*Can_rxData*), making it accessible to the rest of the system for further processing. Additionally, the function sets the *Can_bRxFlag*

flag to *true* to notify the main loop that new data is available and needs to be passed on to the relevant peripherals.

LIN Callback Function

The *LinIf_RxIndication* function is called everytime a new LIN frame has been received and can be copied from the receive buffer for further processing. This happens when the S32K144 provides updated data from the proximity sensors. Since this update affects the visual output, the function sets the *display_refresh* flag to *true* to initiate a screen refresh. Additionally, it checks whether any of the sensors report the value *0x00*, which corresponds to *ZONE_A*, the closest proximity zone. If such a value is detected, the function also activates the flag for the brake lights.

XS2410 Control Function

The *XS2410_updateOnReceivedData* function checks all the relevant flags that were set beforehand in the current main loop iteration and performs the necessary actions accordingly. It enables or disables the appropriate channels by sending SPI frames to the XS2410, and if turn indicators or hazard lights are involved, it also starts the blinking timer to handle their flashing behavior. The blinking timer notification callback is the function *Lights_Blink*.

Triggering XS2410 Watchdog Failure and Repair

Another feature of this device is the ability to intentionally trigger a watchdog failure condition upon user request. This is implemented by the *XS2410_triggerWdgFail* function, which works by stopping the periodic timer responsible for feeding the XS2410 watchdog. As a result, once the watchdog window expires, the HSS enters an error state and all LEDs are turned on due to the LIMP mode configuration. Additionally, the device's FAULTB pin is pulled low to signal the failure to the controlling board. The periodic LED refresh function monitors the state of this pin to visually indicate whether a communication failure is occurring.

If the HSS watchdog failure button is released, the board initiates a recovery process to restore communication with the HSS. This includes reapplying the settings from before the failure and resuming the periodic watchdog refreshes, thereby returning the device to its normal operating mode. The function *XS2410_repairDevice* implements this behaviour.

Trunk Opening Mechanism

The implementation of the trunk opening mechanism is fundamentally simple—it involves sending an SPI frame to the HB2001, instructing it to rotate the motor either to the right, to the left, or to stop. The board determines when to stop the motor based on interrupts received from pins that monitor the position of the hinge, indicating whether it is fully open or closed. The interrupt handlers are *HB2001_notificationClosed* and *HB2001_notificationOpened*. The state of these pins is also monitored in order to prevent the opening/closing action when there is the opposite one currently ongoing.

LED Timer

The LED timer on this board provides the same functionality as the one used for the S32K144, but with a greater number of communication lines to visualize. This board

manages three separate LED visualization paths: one for the LIN communication with the S32K144, and two for the SPI communications with the XS2410 and HB2001 peripherals. Each communication path is associated with a dedicated flag that indicates whether the communication is currently active. These flags are stored within *commVisState_t* structures (*XS2410State*, *HB2001State* and *LinState*) and are checked each time the LED periodic timer expires. Additionally, since a watchdog failure can be triggered during communication with the XS2410, it is also possible to visualize it — the LED path will start blinking red. The presence of this error is detected by checking the state of the FAULTB pin.

4.2.3 S32K312 Input Peripheral Firmware

This section describes the firmware functionality for the second S32K312 board. All related files are contained within the *S32K312_Demo_2* project.

Initialization

This project uses the same components as the previous S32K312 project, so the used components remain very similar, with the addition of the Sbc_FS26 and Wdg_FS26 driver. This RTD driver is used to initialize and control the FS26 system basis chip. This initialization part also includes the setup sequence for the CD1020 device.

Main Loop

Algorithm 3 The main loop of the second S32K312 board firmware.

```
1:  display_refresh ← false
2:  data_modified ← false
3:  while true do
4:      if display_refresh == true then
5:          Display_Refresh(LIN_DATA)
6:          display_refresh ← false
7:      end if
8:      if data_modified == true then
9:          CAN_Send(Can_txData)
10:     end if
11:     if switch_status[SBC_WDG_FAIL] == on & fs26_wdg_running then
12:         FS26_triggerWdgFail()
13:     end if
14:     if switch_status[SBC_WDG_FAIL] == off & !fs26_wdg_running then
15:         FS26_triggerWdgRepair()
16:     end if
17: end while
```

The code again features an endless loop that continuously checks whether any flags were set during execution, either when new input data becomes available from the CD1020 switch detector or from the LIN transceiver.

The first part checks the *display_refresh* flag, which, as in the previous example, signals the main loop to refresh the data currently being displayed. The next part checks the *data_modified* flag, which indicates that new data from the CD1020 has been stored in the

corresponding buffer and can be sent over CAN to the second S32K312 board. Lastly, the state of the SBC_WDG_FAIL button is determined based on the most recently received data from the CD1020. If the button is pressed, the watchdog turn-off functionality is initiated. If it is released, normal watchdog operation is restored.

LIN Callback Function

The LIN callback function (*LinIf_RxIndication*) serves the same purpose as in the previous S32K312_1 code: setting the *display_refresh* flag, updating the proximity sensor buffer data, and checking whether the brake lights need to be turned on.

CD1020 Interrupt Handler

The device is configured to trigger an interrupt every time the INT_B pin is pulled low. The handler for this interrupt is *CD1020_Notification*, which performs an SPI read on the register that stores the current state of all the pins. Inside the handler, the flag *data_modified* is also set to *true*, allowing the rest of the application to detect that the data has changed and requires attention.

FS26 Watchdog Failure and Restoration

The continuity of communication with the FS26 SBC is monitored by a watchdog. This watchdog can be deliberately interrupted by pressing a pushbutton, which consequently stops the S32K312 from sending periodic watchdog refreshes. This error state is recognized by the firmware when the INT_B pin of the FS26 is pulled low. The error condition persists until the button is released, at which point normal operation of the device is restored. While the device is in this error state, all data from the CD1020 and S32K144 is ignored, and no data can be transmitted to the other S32K312 board over CAN. The function implementing the watchdog failure is *FS26_triggerWdgFail* and the function restoring the normal operation is *FS26_triggerWdgRepair*.

LED Timer

Similarly to the other firmwares, this one also handles its own LED paths—specifically, the CAN and LIN communication paths, as well as the SPI paths from the FS26 and CD1020 devices. All of this communication remains functional as long as the FS26 device is operating correctly. That's why, if an error occurs in communication with the FS26, all other communication channels also report a fault state. Instead of lighting up in their individual protocol colors, all LED paths will blink red.

4.3 Physical Model

The next step was the creation of the actual physical model. I selected an RC kit Ford Mustang 1968 in a scale of 1:10 for the car model, which is small enough to still be considered portable and spacious enough for a clear visualization and housing of all the components. The body is made of polycarbonate resin, a material known for its durability and flexibility, making it well-suited not only for the demanding conditions of RC racing but also for this demonstration platform.

4.3.1 Base Model

All the panels for the base of the physical model were designed using FreeCAD[2]. The individual parts were then custom-cut from white 3mm plexiglass with the assistance of PlasticExpress[3]. Plexiglass was chosen not only for its durability but also for its aesthetic appeal, which made it suitable for this application.

The top platform (Figure 4.4) contains cut outs for the LED paths. These paths are covered with a light dimming foil as to refract the light coming from the LEDs and make it shine more evenly along the paths.

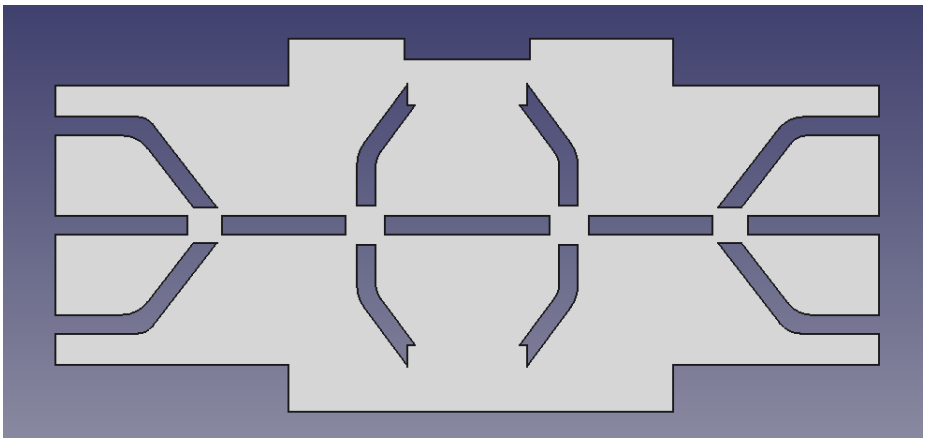


Figure 4.4: 3D model of the top platform.

Additionally, descriptive stickers pictured in image 4.5 and 4.6 were printed and placed on both the control panel and the base panel to improve overall clarity and make the system more understandable for users.

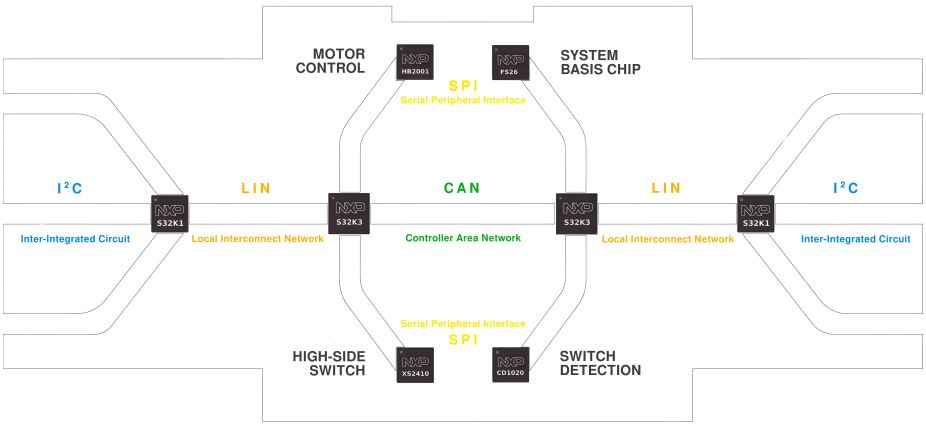


Figure 4.5: Descriptive sticker for the base panel.

[2]https://www.freecad.org/
[3]https://plasticexpress.cz/

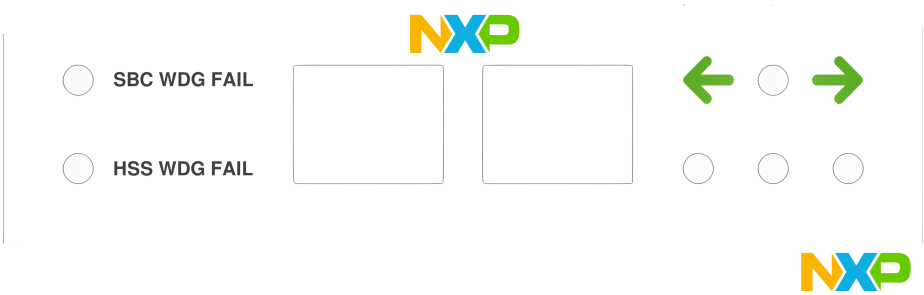


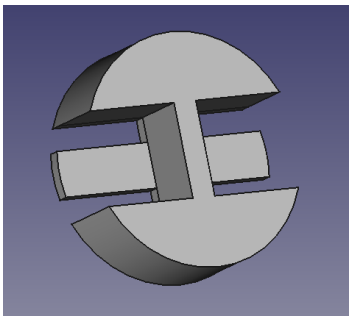
Figure 4.6: Descriptive sticker for the control panel.

4.3.2 Component Attachments

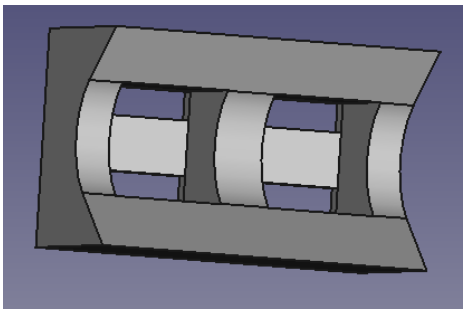
All parts were modeled in FreeCAD and 3D printed using white PETG filament.

Lights Attachment

The headlights, turn indicator lights and brake lights mounts, as seen in Figure 4.7a and Figure 4.7b are designed to be attached to the corresponding areas of the car model, replicating the placement of lights in a real vehicle.



(a) 3D model of the front lights mount.



(b) 3D model of the back lights mount.

Parking Sensors Attachments

The mount shown in Figure 4.8 is designed to securely attach six parking sensors along the base of the car, ensuring they remain upright and properly aligned with the base platform.

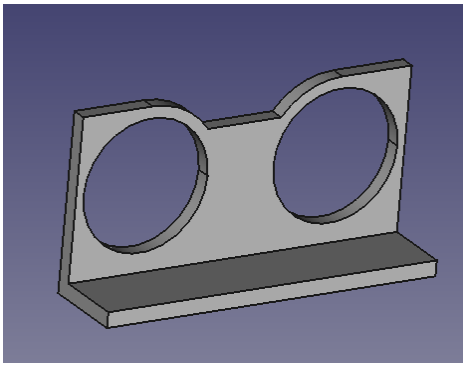


Figure 4.8: 3D model of the parking sensor mounts.

Trunk Opening Mechanism Hinge

The mechanism in Figure 4.9 will be attached to the motor shaft and used to lift the car model from the platform. Additionally, it will reinforce the side of the car model, preventing any bending or deformation during this operation.

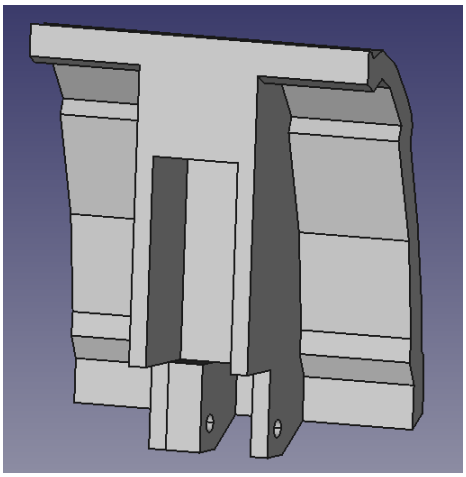


Figure 4.9: 3D model of the hinge the car model will be attached with.

4.3.3 Final Assembly

For assembly, a special adhesive for hard plastics was used to ensure strong and durable bonds. Additionally, support reinforcements were 3D-printed and glued to the edges of the housing, making it more resistant to breakage. The final product is durable but easy to disassemble at the same time, in case some future modifications need to be made.

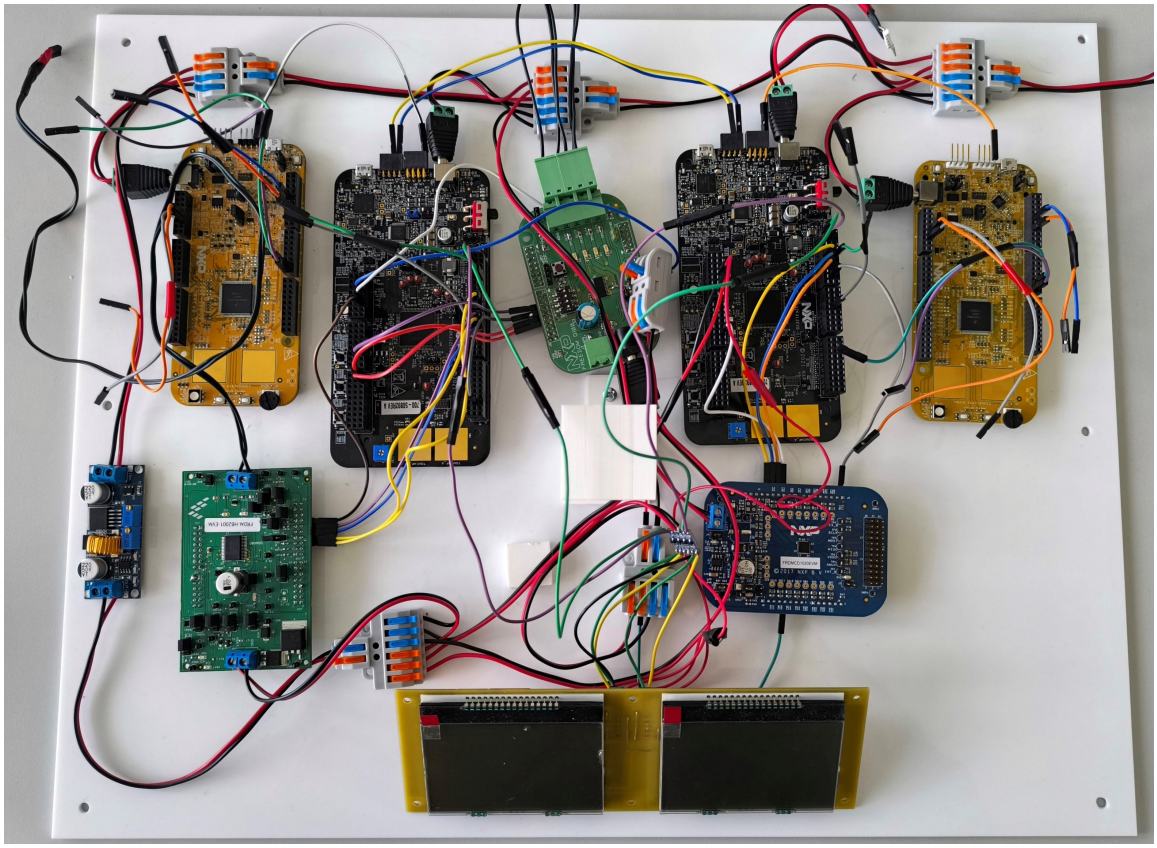


Figure 4.10: The inside of the platform.

Image 4.10 shows the bottom part of the platform containing all the MCUs and peripherals. All of these are enclosed and therefore not visible for the user. Image 4.11 shows the demonstration platform completely assembled.

Figure 4.11: The completed demonstration platform.

Chapter 5

Evaluation and Future Improvements

This chapter evaluates the completed demonstration platform in terms of both its functionality and its intended purpose. Initial testing was conducted by me, the developer, to ensure that all features operated as expected. Once the system was verified, it was presented to other users to assess its usability and the clarity of the project. Additionally, this chapter proposes several ideas for future improvements and extensions that could enhance the platform's capabilities and educational value.

5.1 Testing

The finished project went through a couple of tests to make sure everything worked as expected. These tests focused on the individual functionalities of the system, but always within the context of the complete setup. Features like ECU communication, LED signaling, user input handling, and actuator control were each tested to ensure they functioned correctly as part of the overall system. A logic analyzer was used to monitor the communication interfaces and verify that messages were correctly transmitted and received. Visual inspection was used to check LED behavior and actuator responses, while user input functions were tested manually through button presses.

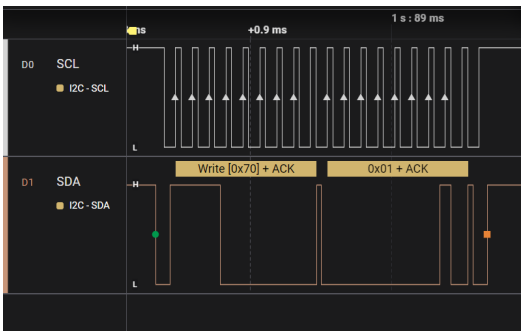
5.1.1 Testing Scenarios

This section outlines the individual testing scenarios, detailing the expected outcomes and the actual results of each test. While the images only show a single illuminated dot along the pathway to represent the data flow, in reality the entire path lights up sequentially as the light travels.

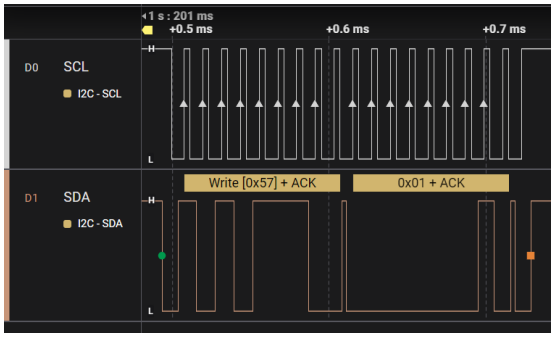
Proximity Sensor Visualization

To verify that the proximity sensor visualization is functioning correctly, it is sufficient to place an object in front of the sensors and observe whether the image on the display changes. For more precise testing, I used a ruler to check if the measurements corresponded to the correct highlighted zones on the display. The object that should be detected was placed 15 centimeters away from the sensor on the left side in the back of the car.

The S32K144 is expected to forward the proximity zone values received from the sensors—0x02 (ZONE_C), 0x03 (ZONE_NONE), and 0x03 (ZONE_NONE)—to the S32K312 via the LIN interface. The LED paths leading from the proximity sensors to the S32K144 are expected to light up in blue, indicating I^2C communication. Additionally, the LED path between the corresponding S32K144 and S32K312 should light up in orange, representing LIN communication, as the sensor data is transmitted for visualization.



(a) Setting the I^2C multiplex address.



(b) Sending a read command to the chosen sensor.

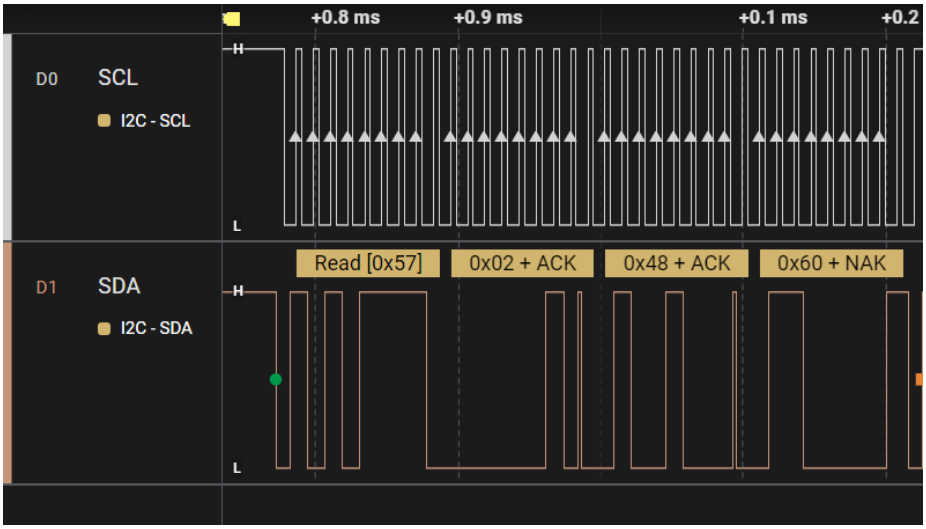


Figure 5.2: Read response from the sensor.

Figures 5.1a, 5.1b, and 5.2 show the logic analyzer capture of the communication between the S32K144 and the proximity sensor during a read operation, as explained in 4.2.1. The first step involves setting the multiplexer address to select the desired sensor, in this case, sensor 1 (address 0x01). Next, a read command is issued to the sensor at address 0x57. Finally, the response from the sensor is read. Applying equation 4.1 to the received data results in a value of 14.96, which corresponds closely to the expected distance measurement of 15 cm.

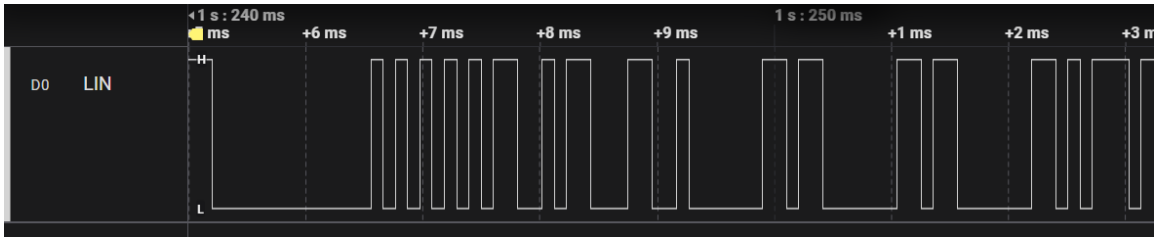


Figure 5.3: LIN frame being sent from S32K144 to S32K312.

Image 5.3 shows the analyzer capture of the LIN frame transmitted from the S32K144 to the S32K312, containing the sensor data. At the begging of the frame the sync and break field are visible, followed by the identifier. Then three data bytes-0x02, 0x03, 0x03 are transmitted, followed by the checksum field.

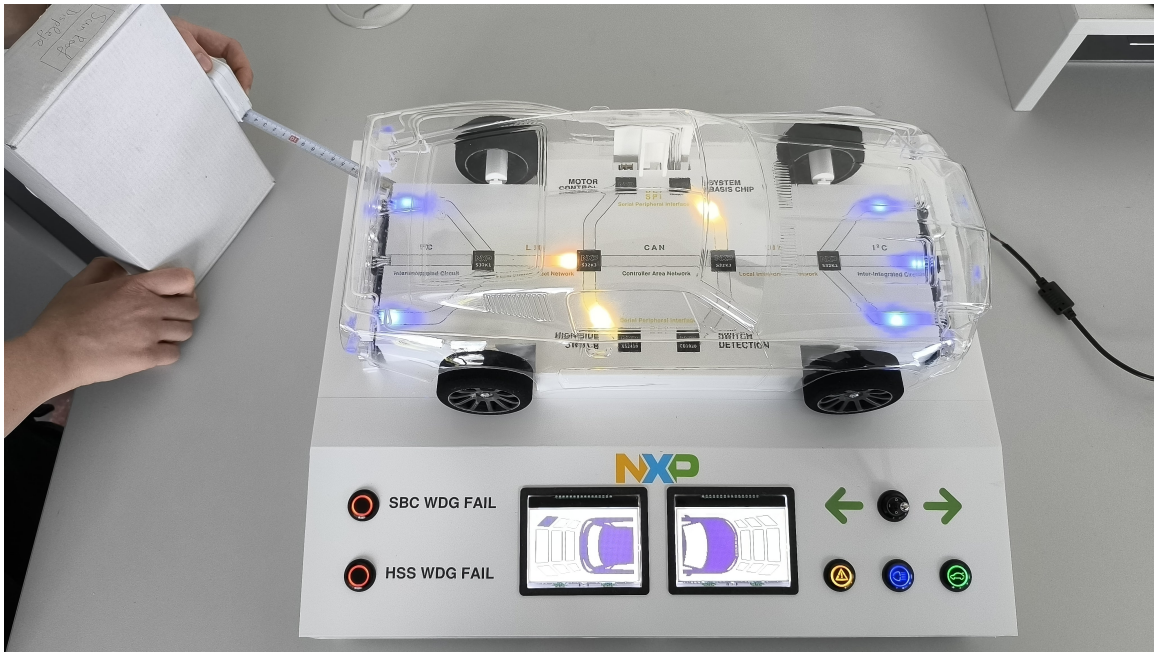


Figure 5.4: Object 15cm away from the left rear sensor.

As visible in image 5.4, the resulting state is as expected.

Turning on the Headlights

The headlights should turn on when the headlight button is pressed and turn off when it is released. Simultaneously, the LED path from the CD1020 to the first S32K312 should light up in yellow, indicating SPI communication. The same should occur for the LED path between the two S32K312 boards, where data is transmitted over CAN (green), and then from the second S32K312 to the XS2410, where another SPI-related yellow LED path lights up. The same sequence of events should occur during the turn-off process as well. The SPI path to XS2410 should keep blinking even after they the lights are on, because the board keeps refreshing the watchdog.

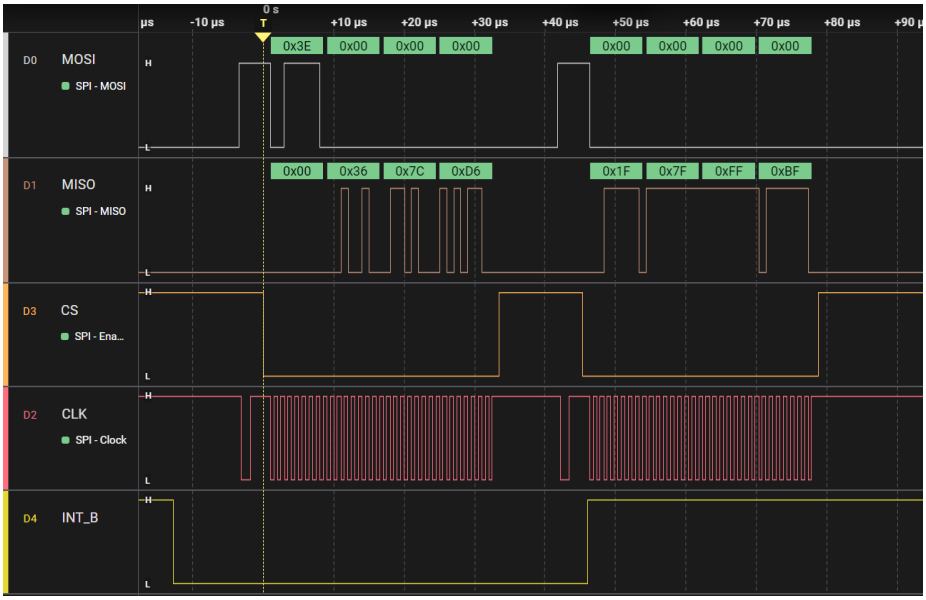


Figure 5.5: SPI command being sent from CD1020 to S32K312 with switch status.

Image 5.5 displays a logic analyzer capture of the moment when a button is pressed, causing the INT_B pin of the CD1020 to be pulled low, indicating a status change. In response, the S32K312 microcontroller issues an SPI read command to address 0x3E [10]. The CD1020 replies with the switch status, where the SG6 bit is read as zero, indicating that the SG6 input—routed to the headlights button—has been activated. This confirms that the button is engaged.

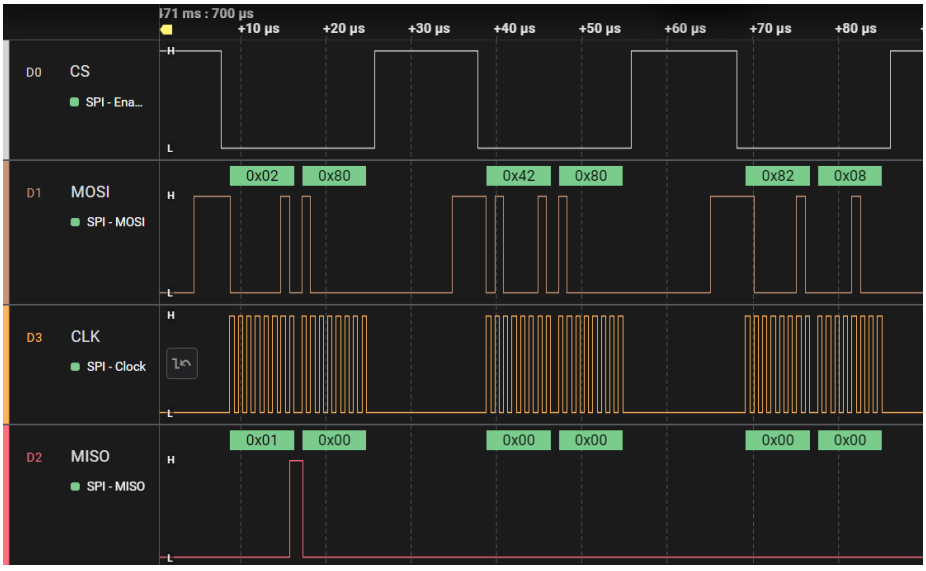


Figure 5.6: SPI command being sent from S32K312 to XS2410 to turn on the lights channel.

Image 5.6 illustrates the command sequence sent to the XS2410 over SPI to activate the lights output channel (OUT4). The process consists of two main steps. First, the current state of the output channels is read, as shown by the initial two SPI frames. The retrieved output status—in this case, all zeros—is then modified to enable the desired channel. To

activate channel 4 (corresponding to the bit value 0x08), this value is written back to the output register located at address 0x02. On the SPI analyzer, the value 0x82 indicates a write operation to address 0x02 [9].

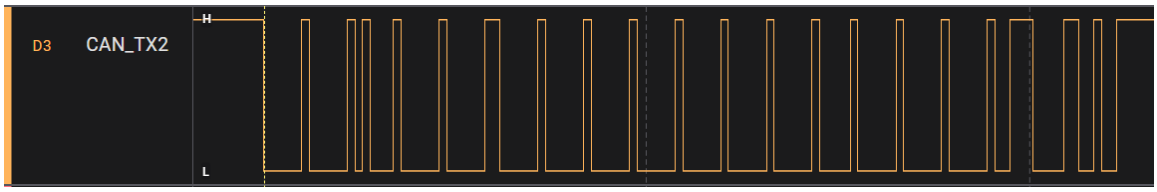


Figure 5.7: CAN frame being sent between the two S32K312 boards.

Image 5.7 captures the CAN frame being transmitted between the two S32K312 boards, containing a payload of eight bytes. All bytes are set to 0x00 except for byte number five, which holds the value 0x01. This specific byte represents the headlights flag, and the value 0x01 serves as a command to activate the headlights.

Figure 5.8: Headlights button pressed.

Image 5.8 shows that the system is behaving as expected, with all the LED paths lighting up accordingly.

Turning on the Warning Lights

Similarly to the previous test case, the warning lights are triggered by pressing the pushbutton with the warning sign. When activated, both turn indicators should blink at a constant rate, and when the button is released, the blinking should stop and the lights should remain off. The LED paths should light up in the same pattern as in the previous test case.

Figure 5.9: Warn lights button pressed.

Although the blinking of the lights is not visible in the static image, image 5.9 shows that both lights are on, as well as all the corresponding LED paths.

Turning on the Turn Indicator Lights

Flipping the three-state switch to the left or right relative to the default position should activate the corresponding turn indicator, which will blink at a constant rate. The LED path light-up sequence remains the same as in the previous two test cases.

Figure 5.10: Right turn indicator.

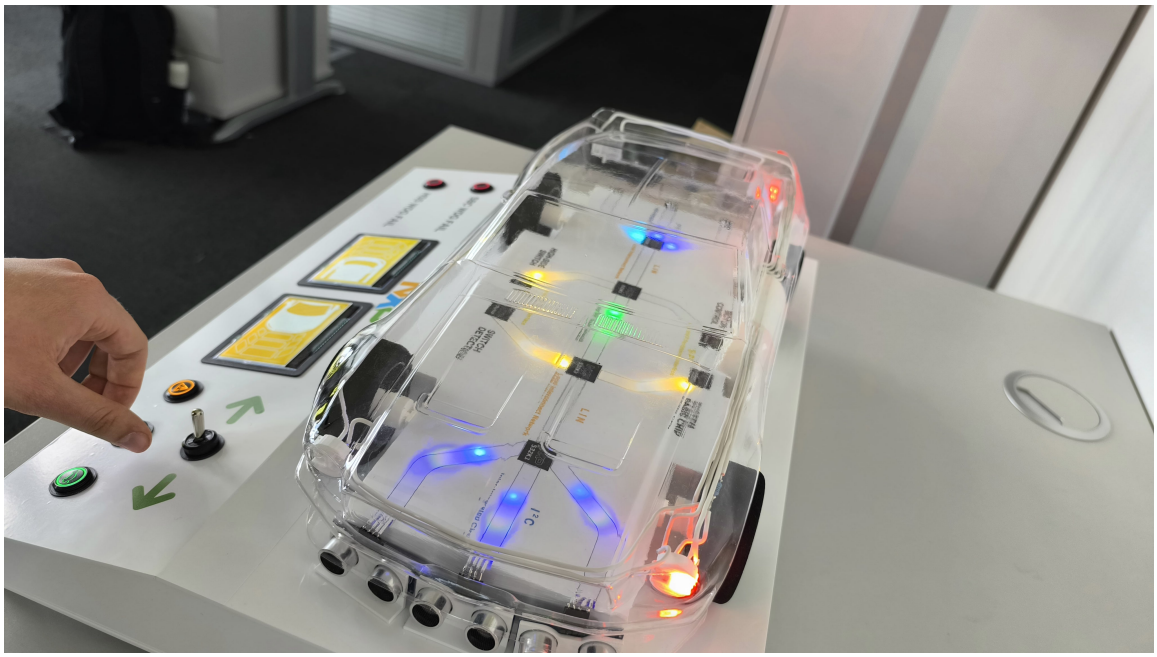


Figure 5.11: Left turn indicator.

Once again, the blinking effect is not visible in the static images 5.10 and 5.11, but the system behaves as expected.

Triggering the Brake Lights Functionality

Turning on the brake lights requires a few additional steps. They are activated when an object is detected too close to the car—specifically, if any of the proximity sensors reports

a *ZONE_A* reading. To trigger this condition, an object must be placed in close proximity to a sensor. As a result, the brake lights should illuminate red, and once the object is removed, they should turn off again. The LED paths that should light up include the I^2C path from the sensors to the S32K144, the LIN path from the S32K144 to the S32K312, and the SPI path from the S32K312 to the XS2410. Additionally, if the signal originates from the S32K312 that is not directly connected to the XS2410, the information must be transmitted over CAN to the second S32K312.

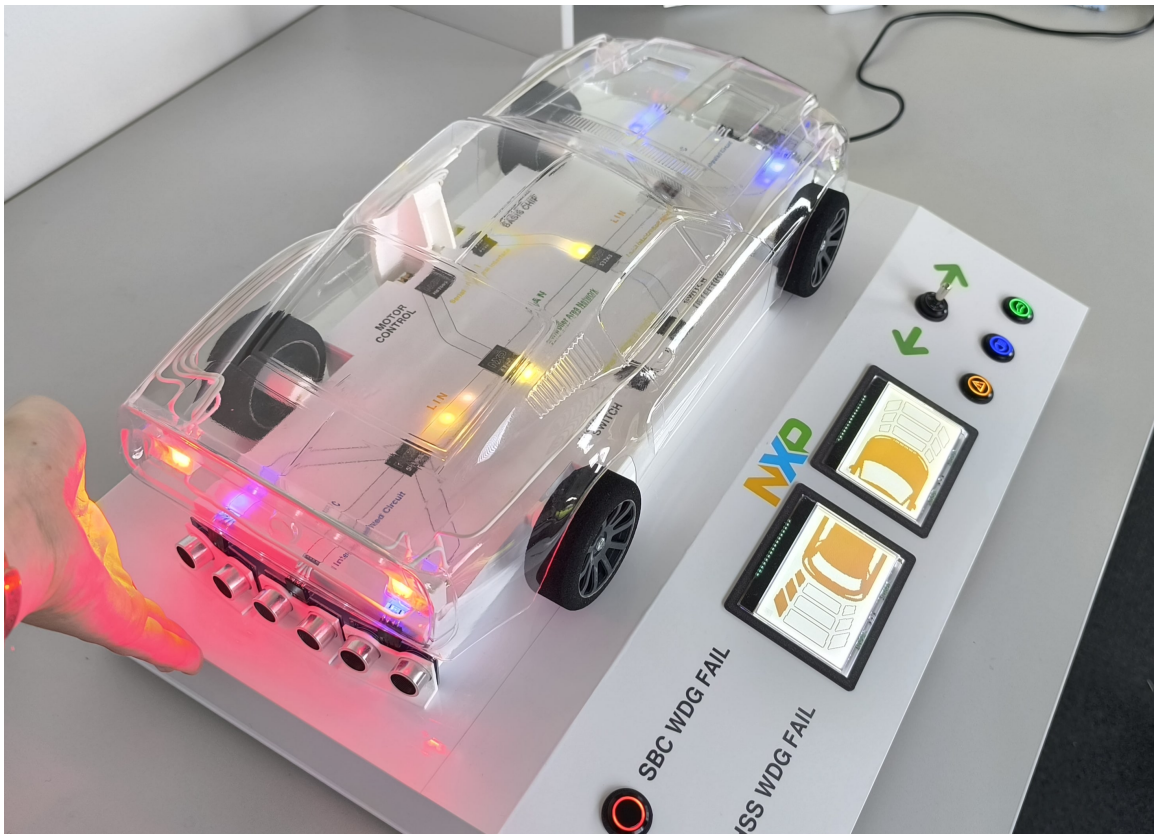


Figure 5.12: Object in the closest zone in the back of the car causes brake lights to turn on.

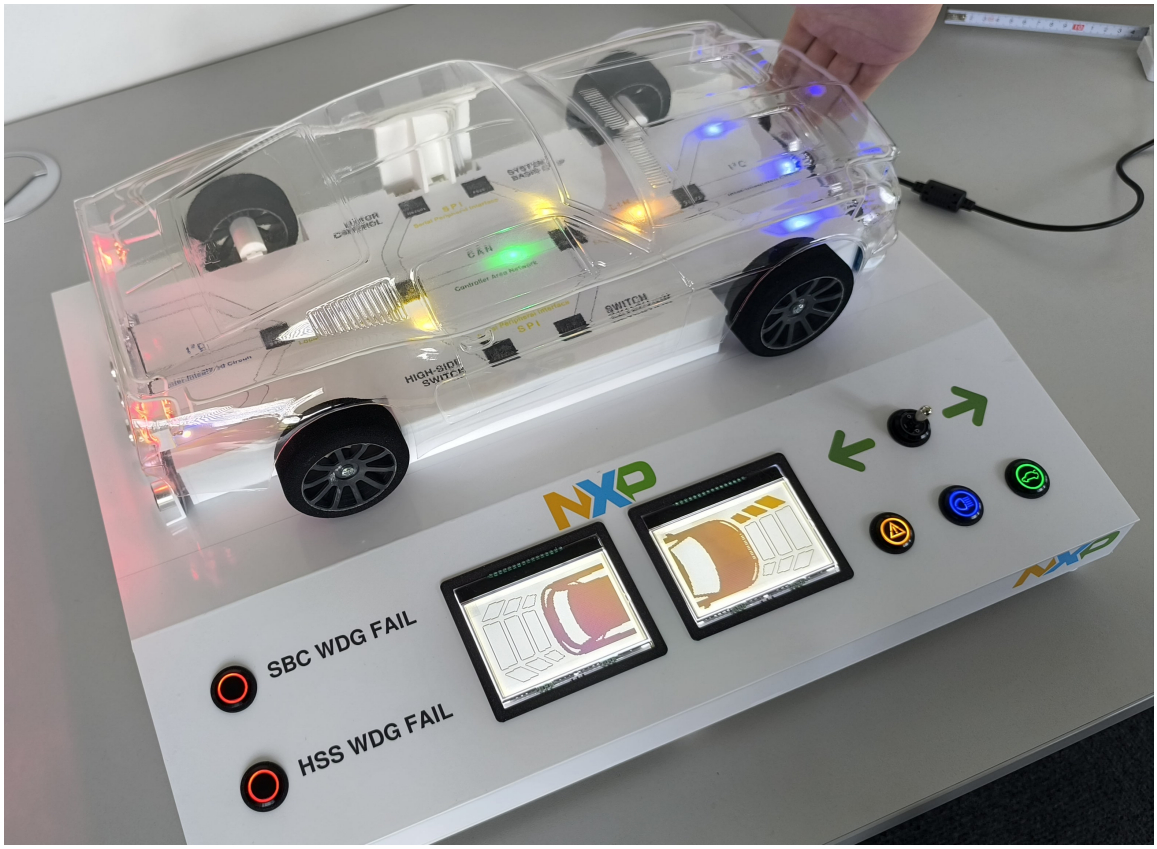


Figure 5.13: Object in the closest zone in the front of the car causes brake lights to turn on.

Image 5.12 shows that the brake lights are on, along with the corresponding LED paths when the object is placed in the back of the car. Image 5.13 illustrates the case when the object is on the opposite side of the car, where the additional CAN path is also lit up.

Triggering the Trunk Opening Mechanism

Another functionality triggered by pressing a button is the lifting of the case to expose the LED paths inside. This is done by pressing the pushbutton with the trunk icon. If the case is currently closed, it should begin opening; if it is open, it should start closing. If it is in the middle of either action, the system will first complete the current motion. The LED paths light up as follows: first, the button press is communicated via the yellow SPI path from the CD1020 to the first S32K312. This board then sends the information over the green CAN path to the second S32K312, which initiates the motor motion by sending an SPI command to the HB2001. Once the action is complete, the path from the HB2001 to the S32K312 lights up again, indicating the command to stop the motor.

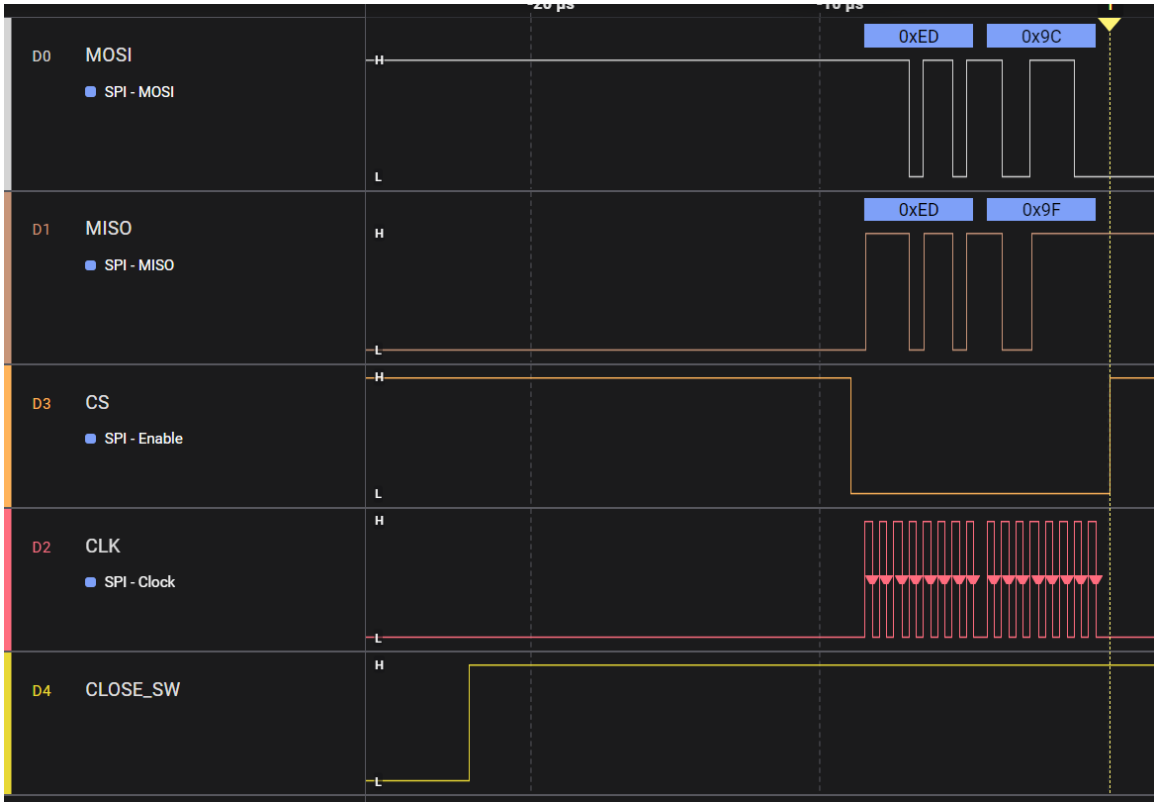


Figure 5.14: Reception of the closing button engaged and stopping the motor.

Image 5.14 illustrates the analyzer capture of the enforced closing switch event, followed by an SPI command sent to the HB2001 to stop the motor. This command transmits the frame with value 0xED9C, corresponding to a write operation to control register 0x60, setting both VIN1 and VIN2 bits to zero, thereby disabling the motor drive inputs [6].

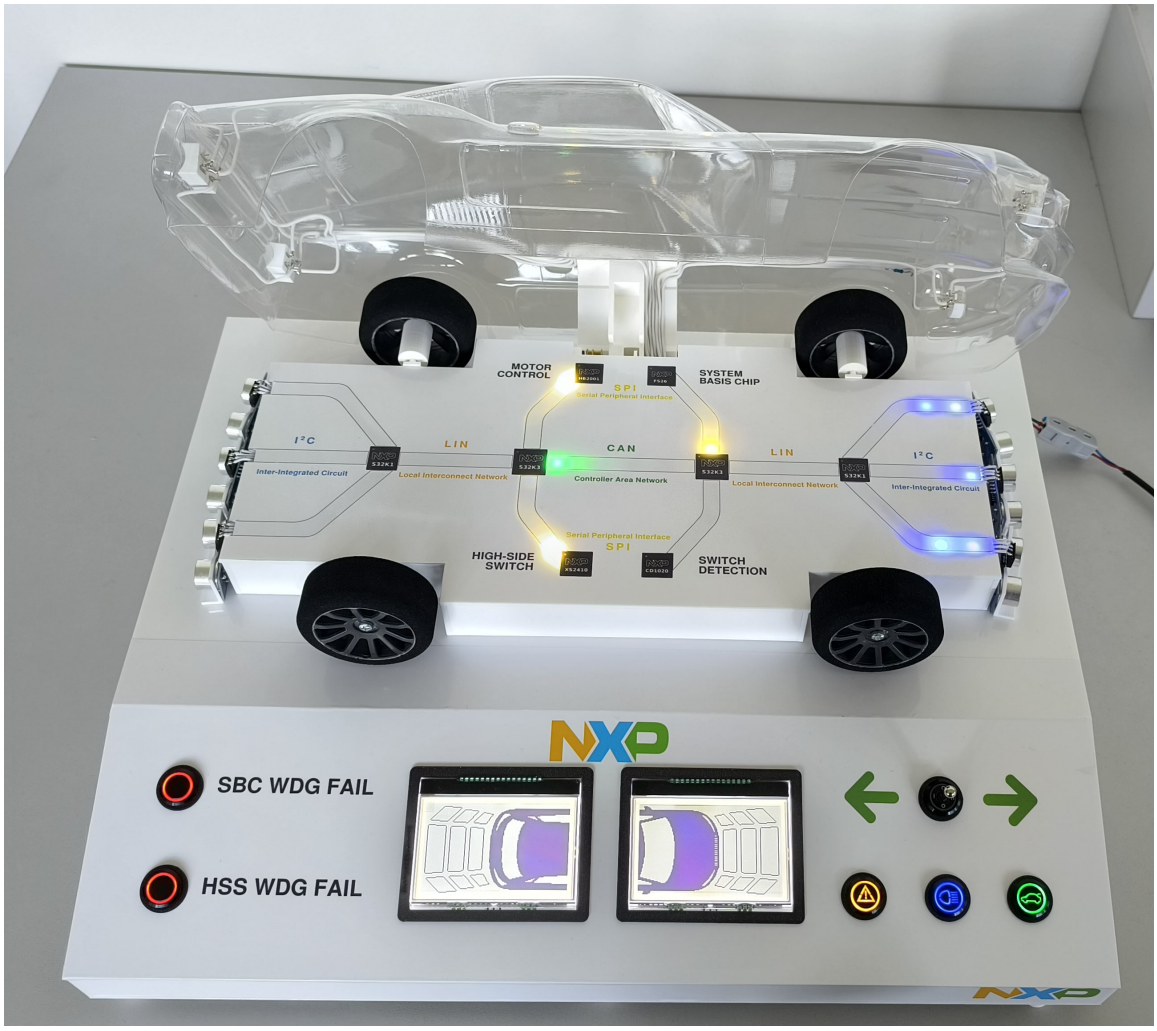


Figure 5.15: Opened trunk starting the closing action.

Images 5.15 shows the state when the trunk is fully open and starts to close again upon the pushbutton press. Both the CAN path and the Motor Control path are lit up.

Triggering the XS2410 Watchdog Failure

The XS2410 watchdog failure is triggered by pressing the lower red button on the control panel. When the button is not pressed, the LED path between the XS2410 and the S32K312 lights up periodically to indicate that the watchdog is being refreshed. Once the button is pressed, the refresh signal stops, and the LED path remains off until the watchdog timeout window expires. After the timeout, the XS2410 reports the fault by pulling the INT_B pin high, at which point the LED path starts blinking red to indicate an error. Simultaneously, all vehicle lights turn on, representing the XS2410's configured LIMP state. This fault condition remains active as long as the button is pressed. Once it is released, the XS2410 resumes normal operation, and the LED path begins blinking periodically in yellow (SPI) again, signaling successful watchdog refreshing.

Figure 5.16: XS2410 Watchdog interrupted.

Image 5.16 shows the error state of the LED communication path between XS2410 and S32K312, when the pushbutton was released, the system returned to normal operation.

Triggering the FS26 Watchdog Failure

The final functionality that the user can initiate is triggering the FS26 watchdog failure by pressing the upper red button on the control panel. Similar to the XS2410 watchdog behavior, the LED path between the FS26 and the S32K312 blinks periodically during normal operation to indicate ongoing watchdog refreshing. When the button is pressed and the watchdog timeout period elapses, the FS26 enters an error state and reports the fault to the S32K312. As a result, the LED path begins blinking red to signal the error condition. Additionally, all other LED paths connected to the S32K312 will also blink red, reflecting the FS26's critical role in powering and enabling system-wide communication. Once the button is released, the FS26 exits the fault state, and the system resumes normal operation, with all LED paths returning to their standard behavior. MCUs not directly impacted by the FS26 failure remain fully operational, which means the proximity sensors continue collecting data and the other S32K312 board will continue refreshing the HSS watchdog.

Figure 5.17: FS26 Watchdog interrupted.

Image 5.17 shows the system's response to triggering the FS26 watchdog timeout, which behaves as expected. After the button was released, the system successfully returned to its default operational state.

5.1.2 User Feedback

Next, I wanted to ensure that the entire project is understandable for users without a technical background. Since my target audience consisted of people with little to no prior technical knowledge, I selected a small group of friends and family members as the test subjects. I asked them to trigger the same scenarios described in the previous section. These scenarios were successfully executed, further confirming that the chosen design is user-friendly.

5.2 Possibilities of further development

Given the vastness of the automotive industry, it is impossible to cover every aspect within the scope of this project. However, there are many directions in which this work could be expanded if future development is desired. Some potential improvements could include:

- **Analyzer test points** Adding accessible test points for each communication route would allow the use of a logic analyzer to capture and inspect the data being transmitted over the wires.
- **Support for Additional Automotive Communication Protocols** While this project is relatively small in scale, incorporating other commonly used protocols such as FlexRay or Automotive Ethernet could significantly increase its educational and practical value.

- **Expanded Simulation Scenarios** The range of possible automotive situations is broad, and the project could be extended to simulate additional real-world scenarios.
- **FlexGUI Integration** Integrate the project with FlexGUI[1], so it would be possible to operate and modify from a computer.

[1]FlexGUI is another NXP product - https://www.nxp.com/design/design-center/software/analog-expert-software-and-tools/flexgui-software-tool-for-evaluation-of-reference-design-kits:FLEXGUI-SW.

Chapter 6

Conclusion

This thesis presented the development and implementation of a demonstrative platform for automotive communication protocols, providing an interactive and visual approach to understanding the data exchange between Electronic Control Units (ECUs). The physical model successfully simulates key automotive subsystems, such as lighting control, parking assistance, and trunk operation, while employing CAN, LIN, SPI, and I2C protocols to mimic real-world automotive communication.

By integrating visual feedback through LED paths, the platform enhances the learning experience, making otherwise abstract communication concepts more tangible. The modular and scalable nature of the design allows for potential future expansions, such as incorporating additional automotive protocols like FlexRay or Ethernet, introducing new simulation scenarios, or adding test points for signal analysis.

Overall, the project achieved its goal of bridging the gap between theoretical knowledge and practical understanding of automotive communication. It serves as a valuable educational tool for students and professionals seeking insight into the operation of modern vehicle networks. Further enhancements could include software integration for remote monitoring and interaction, making the platform even more versatile for training and demonstration purposes.

Bibliography

[1] *Serial Peripheral Bus* online. Real Digital. Available at: https://www.realdigital.org/doc/6c55fef7bba4a22ff35dce46a3c359af. [cit. January 12, 2025].

[2] *CAN Specification* online. Version 2.0. ROBERT BOSCH GmbH, 1991. Available at: http://esd.cs.ucr.edu/webres/can20.pdf. [cit. March 22, 2025].

[3] *LIN Specification Package* online. Revision 2.2A. LIN Consortium, 2010. Available at: https://www.lin-cia.org/fileadmin/microsites/lin-cia.org/resources/documents/LIN_2.2A.pdf. [cit. March 22, 2025].

[4] *TCA9548A Low-Voltage 8-Channel I2C Switch with Reset* online. Texas Instruments, 2012. Available at: https://www.ti.com/lit/ds/symlink/tca9548a.pdf. [cit. March 11, 2025].

[5] *Getting Started with the S32K144EVB* online. NXP Semiconductors, 2019. Available at: https://www.nxp.com/document/guide/getting-started-with-the-s32k144evb:NGS-S32K144EVB?section=out-of-the-box. [cit. May 8, 2025].

[6] *MC33HB2001, 10 A H-Bridge, programmable brushed DC motor driver - Data Sheet* online. REV 10.0. NXP Semiconductors, 2020. Available at: https://www.nxp.com/docs/en/data-sheet/MC33HB2001.pdf. [cit. May 8, 2025].

[7] *I2C-bus specification and user manual* online. NXP Semiconductors, 2021. Available at: https://www.nxp.com/docs/en/user-guide/UM10204.pdf. [cit. March 22, 2025].

[8] *Low-Cost Local Interconnect Network (LIN)* online. Microchip, 2023. Available at: https://developerhelp.microchip.com/xwiki/bin/view/applications/lin/. [cit. March 23, 2025].

[9] *MC33XS2410: Quad 100 m / Dual 50 m, 3.0 V to 60 V High-Side Switch Data Sheet* online. REV 8. NXP Semiconductors, 2023. Available at: https://www.nxp.com/docs/en/data-sheet/MC33XS2410.pdf. [cit. May 8, 2025].

[10] *CD1020, 22 Channel Switch Detection Interface Data Sheet* online. REV 5. NXP Semiconductors, 2024. Available at: https://www.nxp.com/docs/en/data-sheet/CD1020.pdf. [cit. May 8, 2025].

[11] *S32K312EVB-Q172 Hardware User Manual* online. REV A3. NXP Semiconductors, 2024. [cit. May 8, 2025].

[12] Corrigan, S. *Introduction to the Controller Area Network (CAN)* online. 2016. Available at: https://www.ti.com/lit/an/sloa101b/sloa101b.pdf?ts=1736256518774&ref_url=https%253A%252F%252Fwww.google.com%252F. [cit. March 9, 2025].

[13] Fürst, S. and Bechter, M. AUTOSAR for Connected and Autonomous Vehicles: The AUTOSAR Adaptive Platform. In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*. 2016, p. 215–217.

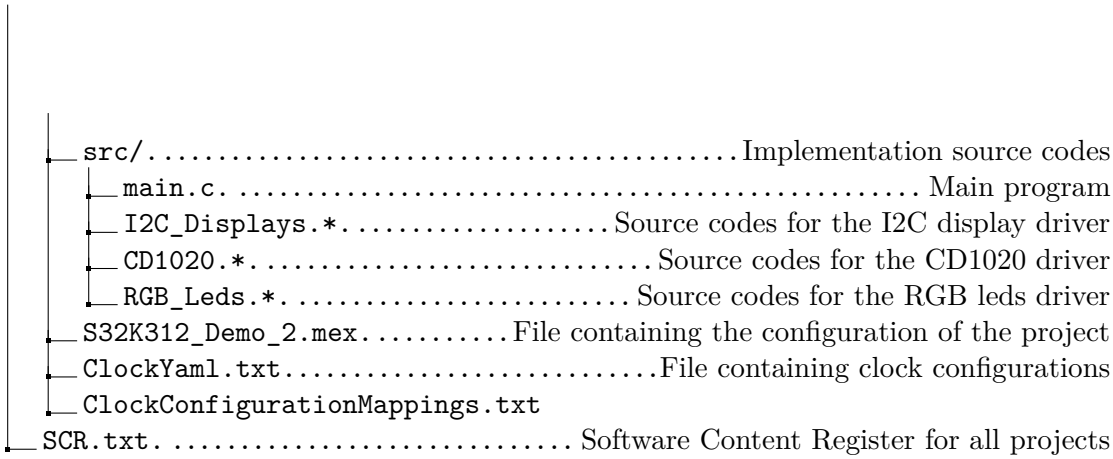
[14] Leens, F. An Introduction to I2C and SPI Protocols. *IEEE Instrumentation Measurement Magazine 2009*, 2009.

[15] Navet, N.; Song, Y.; Simonot Lion, F. and Wilwert, C. Trends in Automotive Communication Systems. *Proceedings of the IEEE*, 2005, vol. 93, no. 6, p. 1204–1223.

Appendix A

Contents of the included storage media

```
/
├── IBT_xkubin27.pdf ..... Pdf version of this thesis
├── print_BP_Kubincova.pdf ..... Printable version of this thesis
├── thesis_src/ ..... Source files for this thesis
├── S32K144_Demo/ ..... Source codes for S32K144 firmware
│   ├── RTD/ ..... RTD drivers generated code
│   ├── board/ ..... Board specific configuration files
│   ├── generate/ ..... Configuration files generated by Config Tool.
│   ├── src/ ..... Implementation source codes
│   │   ├── main.c ..... Main program
│   │   ├── Proximity_Sensors.* ..... Source codes for proximity sensor driver
│   │   └── RGB_Leds.* ..... Source codes for RGB leds driver
│   ├── Configuration.mex ..... File containing the configuration of the project
│   ├── ClockYaml.txt ..... File containing clock configurations
│   └── ClockConfigurationMappings.txt
├── S32K312_Demo_1/ ..... Source codes for the first S32K312 firmware
│   ├── RTD/ ..... RTD drivers generated code
│   ├── board/ ..... Board specific configuration files
│   ├── generate/ ..... Configuration files generated by Config Tool.
│   ├── src/ ..... Implementation source codes
│   │   ├── main.c ..... Main program
│   │   ├── HB2001.* ..... Source codes for HB2001 driver
│   │   ├── I2C_Displays.* ..... Source codes for I2C display driver
│   │   ├── XS2410.* ..... Source codes for XS2410 driver
│   │   └── RGB_Leds.* ..... Source codes for RGB leds driver
│   ├── S32312_Demo_1.mex ..... File containing the configuration of the project
│   ├── ClockYaml.txt ..... File containing clock configurations
│   └── ClockConfigurationMappings.txt
└── S32K312_Demo_2/ ..... Source codes for the second S32K312 firmware
    ├── RTD/ ..... RTD drivers generated code
    ├── board/ ..... Board specific configuration files
    └── generate/ ..... Configuration files generated by Config Tool.
```


src/ Implementation source codes
main.c Main program
I2C_Displays.* Source codes for the I2C display driver
CD1020.* Source codes for the CD1020 driver
RGB_Leds.* Source codes for the RGB leds driver
S32K312_Demo_2.mex File containing the configuration of the project
ClockYaml.txt File containing clock configurations
ClockConfigurationMappings.txt
SCR.txt Software Content Register for all projects