Dzambic Maid, Bsc

Development and Integration of a Rapid Prototyping System Utilising XCP Protocol

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Electrical Engineering

submitted to

Graz University of Technology

Supervisor

Dipl.-Ing. Dr.techn. Georg Macher

Institute for Technical Informatics
Head: Univ.-Prof. Dipl.-Inform. Dr.sc.ETH. Kay Römer

Graz, September 2020

Statuary Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Acknowledgement

The work that is culminating with this thesis was carried out at the Institute of Technical Informatics at the Technical University of Graz and in cooperation with AVL List GmbH.

At first, I would like to express my gratitude and appreciation to Dr. Omar Veledar from AVL List for the opportunity to cooperate with AVL List and for great support and guidance throughout the whole thesis process.

Furthermore, I would like to thank DI Christoph Kreuzberger from AVL List for great technical support and continual provision of useful directions to overcome encountered technical issues.

I also wish to thank my supervisor Dipl.-Ing. Dr.techn. Georg Macher from the Institute of Technical Informatics for support and a plethora of useful guidelines and suggestions throughout the whole thesis work.

Last but not least, a big thank you also to my family, girlfriend, and friends, who provided me with great moral support throughout my whole Master's Degree study program.

Abstract

The thesis covers the development of a low-cost, versatile Rapid Prototyping System (RPS) based on a lower-performance micro-controller platform and XCP Calibration Protocol. The system is accompanied by an ”Intelligent Watchdog”, and a gateway for transmitting Ethernet-based XCP messages to the CAN bus and vice-versa. It is applicable to the real-world automotive environment, which demands extensive testing of software that runs on ECUs sourced from a range of providers. The RPS minimizes development delays caused by a lack of costly commercial RPSs. It expands development flexibility and returns an increase in productivity for minor investments.

The application considers usage of the universal calibration protocol XCP and its contribution to the RPS, an intelligent watchdog, and an Ethernet-CAN interface. For this purpose, an XCP-Master controller is developed. An XCP-Slave driver enables access for other calibration tools, such as CANape.

Automotive embedded systems are also introduced in the thesis. The introduction includes specific challenges encountered by embedded automotive software development, rapid prototyping methods, and development tools and processes. As measurement and calibration play a crucial role, the universal calibration protocol XCP and its functionalities are detailed. The utilized CAN and Ethernet communication protocols are briefly presented.

The implementation is evaluated by measurements and validation. The DAQ and Polling measurement modes bypass a function within an external ECU, while the watchdog is evaluated by monitoring a specific variable in the same ECU. The Ethernet-CAN interface is evaluated through bus bandwidth measurements for different data volumes on a standard automotive ECU. A configuration tool is also developed to provide an easy configuration of the RPS platform.

Kurzfassung

Die Arbeit befasst sich mit der Entwicklung eines kostengünstigen, vielseitigen Rapid Prototyping Systems (RPS), dass auf einer Mikrocontroller-Plattform mit geringerer Leistung und dem XCP-Kalibrierungsprotokoll basiert. Das System beinhaltet auch einen „Intelligenten Watchdog“ und einen Gateway zur Übertragung von Ethernet-basierten XCP-Nachrichten an den CAN-Bus und zurück. Weiters ist das System für reale Automobilumgebung anwendbar, in der umfangreiche Tests von Software erforderlich sind, die auf Steuergeräten verschiedener Anbieter ausgeführt wird. Das RPS minimiert Entwicklungsverzögerungen, die durch das Fehlen kostenintensiver kommerzieller RPS verursacht werden. Es erweitert die Entwicklungsflexibilität und bringt eine Produktivitätssteigerung für kleinere Investitionen.

Die Anwendung berücksichtigt die Verwendung des universellen Kalibrierungsprotokolls XCP und dessen Beitrag zum RPS, einem intelligenten Watchdog und einer Ethernet-CAN-Schnittstelle. Zu diesem Zweck wurde ein XCP-Master-Controller entwickelt. Ein XCP-Slave-Treiber ermöglicht den Zugriff für andere Kalibrierungswerkzeuge wie CANape. In der Arbeit werden auch eingebettete Automobilsysteme vorgestellt. Die Einleitung enthält spezifische Herausforderungen für die Entwicklung eingebetteter Automobilsoftware, Rapid Prototyping-Methoden sowie Entwicklungstools und -prozesse. Da Messung und Kalibrierung eine entscheidende Rolle spielen, werden das universelle Kalibrierungsprotokoll XCP und seine Funktionen detailliert beschrieben. Die verwendeten CAN- und Ethernet-Kommunikationsprotokolle werden kurz vorgestellt.

Die Implementierung wird durch Messungen und Validierung bewertet. Die Messmodi DAQ und Polling umgehen dabei eine Funktion innerhalb einer externen ECU, während der Watchdog durch Überwachen einer bestimmten Variablen in derselben ECU ausgewertet wird. Die Ethernet-CAN-

Schnittstelle wird durch Busbandbreitenmessungen für verschiedene Datenmengen auf einer Standard-ECU ausgewertet. Ein Konfigurationstool wurde ebenfalls entwickelt um eine einfache Konfiguration der RPS-Plattform zu ermöglichen.

Contents

List of Figures

1 Introduction

1.1 Motivation and Goal

The development of specialized embedded software is a complex process, limited by constraints unseen by the general-purpose computers. Embedded systems must often rely on limited resources, yet perform complex functions within systems of systems.

Within the automotive domain, the embedded systems face additional safety and security challenges, many of which are resulting from interactions with the outside world. Consequently, the optimal embedded software quality dictates the usage of extensive additional tests and adjustments during its development process. The automotive embedded systems mostly demand testing of software components in a real-world environment, where interaction with other ECUs is possible.

However, as the ECUs are sourced from a range of providers, the timing constraints limit the possibility of directly integrating new software components into the systems. Even minimal changes require several days for integration into the actual ECUs. Rapid Prototyping Systems (RPS) are developed and used to minimize this risk of developmental delays in the automotive domain. The usage of RPSs enables connection to the existing systems to perform measurements and calibration of internal ECU parameters. It also allows bypassing of complete functions, which is beneficial for development and testing. Thus, the RPSs and their ability to rapidly deploy and test software functions are inevitable during the development phase of software components, if the development time and time-to-market are to be minimized.

The available RPSs utilize special ECU interfaces and calibration protocols for direct access to the memory of the target ECU. These extremely powerful systems come at a considerable cost. Consequently, the availability of RPSs is not widespread amongst software function developers. Available RPSs are shared amongst developers, hence not fully utilizing the available human capacity as well as adding to the waiting times and causing prolongation of the development process. Frequently, the functions under test do not even require the high computation power of RPSs, but only their available functionalities. A microcontroller platform, which would have the possibility of accessing other ECUs and performing measurement and calibration, would fully suffice in some cases. Furthermore, utilizing a microcontroller platform as an RPS brings into play real hardware restrictions of ECUs. This gives an insight into how the function to be tested will perform on real hardware and thus discover problems not detectable with professional RPS systems. Having such a system would provide valuable flexibility for the developers and would return an increase in productivity for minor investments.

This triggers the goal of this thesis: deliver an implementation that utilizes the XCP Calibration Protocol (Universal Calibration Protocol) and provides a Rapid Prototyping System with satisfactory functionalities on a lower-performance microcontroller platform. Furthermore, an ”Intelligent Watchdog” functionality will also be implemented, as well as a gateway for transferring XCP messages from an Ethernet-based communication protocol to the CAN bus and the other way around.

Thus, this work will provide tools for automotive embedded system developers in the form of a low-cost and versatile rapid prototyping system, which can pose a satisfactory substitute for scenarios where professional rapid prototyping systems or CAN-adapters are required but are unavailable or when there is a need to closely monitor internal parameters of an ECU. To reach this goal, Infineon’s Aurix TC277 evaluation kit is used as the development platform.

1.2 Objectives

The crucial aspect of the thesis is based on the implementation of an XCP-Master Controller for the target platform and its correct implementation. Besides that, to provide a configurable rapid prototyping system, the target platform must also integrate a configured XCP-Slave driver. To achieve this goal, the following objectives must be accomplished:

- Integration of the XCP-Slave driver on the target platform
- Implementation and integration of the XCP-Master Controller on the target platform
- Utilization of the XCP-Master Controller to provide a rapid prototyping system with the ability to bypass functions in a target ECU
- Utilization of the XCP-Master Controller to provide an ”Intelligent Watchdog”
- Implementation of an Ethernet-CAN Gateway - transferring XCP messages received via Ethernet to the CAN bus and vice versa

1.3 Master Thesis Structure

The thesis is broken into five distinctive chapters, each focusing on a specific aspect of the conducted work. This, introductory, chapter provides the reasoning behind the work and outlines the target of the thesis. The reasoning is further supported by the review of the state of the art, which is presented in chapter 2. That review is also reinforcing the understanding of the fundamentals of automotive embedded systems and communication protocols. It describes the requirements, challenges, and methodologies regarding automotive embedded system development. Furthermore, common automotive bus systems and communication protocols are also described. The main focus is placed upon CAN and Ethernet communication. These are the main communication busses used for this thesis. Additionally, calibration concepts of ECUs are described, with a broad overview of the XCP protocol and its functionalities.

Chapter 3 provides an insight into the conceptualization and the physical implementation that targets the completion of the thesis goals. The chapter also considers the deployed toolchain and the integration and implementation of the XCP protocol for the purposes of utilizing a master and a slave. Furthermore, details about rapid prototyping functionalities, intelligent watchdog, and Ethernet-CAN interface are also described.

Chapter 4 describes and evaluates the physical setup and the measurement campaign. It also provides a representative selection of the measurement results. This chapter includes a description of the test methods and setups for evaluating the RPS, intelligent watchdog, and Ethernet-CAN interface implementations, followed by observation and discussion of the measurement results.

Chapter 5 scrutinizes the findings and yields conclusions based on the evaluation and discussion of the measurement results from chapter 5. Furthermore, it provides an outlook for possible future work based on the outcomes of this thesis.

2 State of the Art

2.1 Embedded Systems

The world is flooded with a vast range of diverse computer systems, ranging from general-purpose computers, such as laptops and tablets, to computer systems, which are ”invisible”, but contribute to a larger system, e.g. vehicles.

Those computer system types are called embedded systems. While general-purpose computers contribute to solutions of a wide range of scientific, technical, economic, and similar challenges, embedded systems are designed to carry out functions of monitoring and control. [7] Therefore, an embedded system could be defined as:

> *a computer system with a dedicated function incorporated into a larger mechanical or electrical system for control of this larger system.[7]*

There are a multitude of important application fields for embedded systems, one of which is automotive electronics.

2.1.1 Embedded Automotive Systems

Embedded automotive systems are responsible for controlling various vehicle processes, such as the operation of the engine, airbags, anti-braking systems, and others. [7] Modern vehicles of today would not be achievable without the thoughtful integration of embedded systems. Their dynamic evolution has been a long and continuous process. In the early stages of automotive electronics, every vehicle function was controlled by a stand-alone

ECU. An ECU is a form of a "mini-computer" which utilizes microcontroller(s), sensors and actuators, in order to carry out a specific set of tasks. [18] However, having a dedicated ECU for every single vehicle function proves to be an inefficient use of resources, while also providing a disjointed overall control. [7] Hence, in recent years, there has been a departure from this development strategy for automotive electronics. With the emergence of ever so more complex functions, there is a need for a new approach, where functions are distributed over several ECUs and are able to communicate with each other. The complexity of modern vehicles can be grasped in the fact that they contain up to 100 ECUs and countless sensors and actuators, as well as up to 5 different bus systems for exchanging up to 2500 signals. [18] In fact, modern vehicles are equipped with more hardware and computing power than the Apollo spaceship was when it flew to the moon. [18] Utilization of embedded systems has made modern vehicles considerably safer and more comfortable to use. The increased effort invested into the development of automotive embedded systems is demonstrated in the anticipated rise of the contribution of the automotive electronics to the vehicle price. The prediction is that the 35 percent contribution from 2010 is to rise to as high as 50 percent in 2030. [10]

As the usage of embedded automotive systems demands their integration into vehicles, these machines with special operational conditions are exposed to a special set of requirements and challenges. Limited resources of ECUs such as memory, timing, and speed, pose a great technical challenge for the design and implementation of automotive-specific applications, especially for safety-critical operation. Software bugs can, in the worst-case scenario, cause life-threatening situations, but also be very costly for the manufacturers because of recalls. Therefore, it is very important to guarantee a high safety factor during all stages of a vehicle's life cycle. [9]

2.1.2 Electronic Control Unit - ECU

For optimal control of the vehicle behavior, ECUs measure large amounts of parameters and include them in the control calculation process. The heart and brain of every ECU is the microcontroller, which is responsible for coordinating processes. A Microcontroller structure is shown in figure 2.1.

In order to control specific processes, ECUs receive electrical sensor signals which are measuring the relevant data. Then, to act in accordance with user expectations and based on the measured values, ECUs determine the behavior of the processes and send triggering signals to actuators, which directly influence the physical processes.

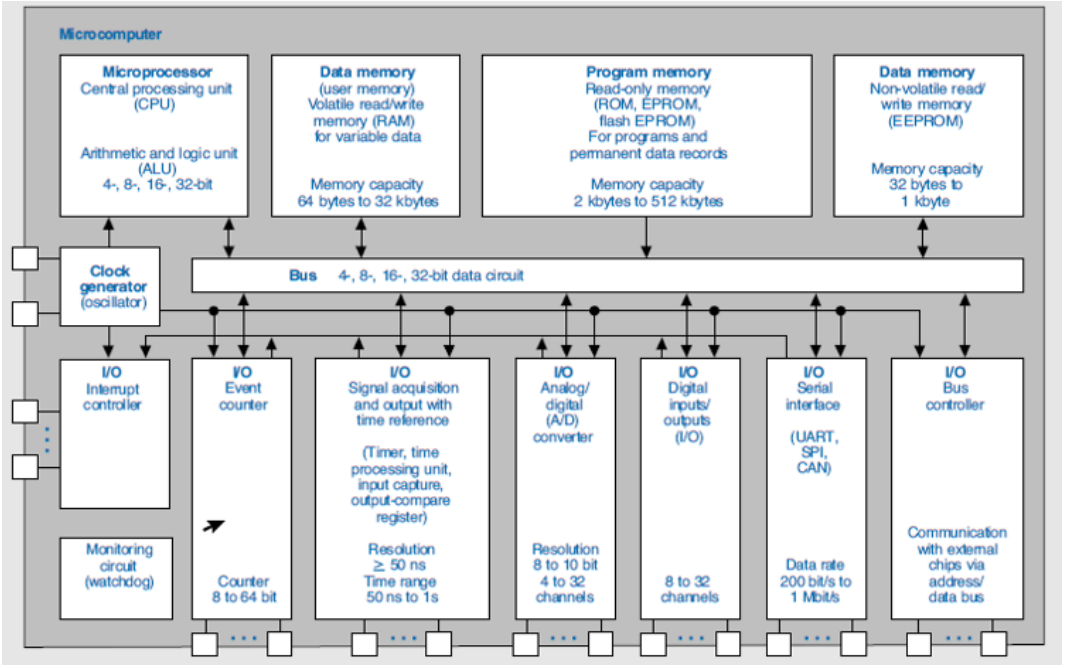

Figure 2.1: Microcontroller architecture[25]

As already mentioned, the fact that ECUs are integrated into the vehicles exposes them to very harsh conditions. Those include[25]:

- Extreme ambient temperatures (from -40 to +65...125 degree C)
- Extremely dynamic temperature changes
- Indirect Materials and supplies (fuel, oil)
- Vibrations and other mechanical stress

Besides that, the ECU operation must not be affected by the level and fluctuations of the battery supply voltage. Since ECUs are also prone to electromagnetic and high-frequency interference, the requirements for Electro-Magnetic Compatibility (EMC) are also very strict. [25]

Besides the microcontroller, ECUs also utilize semiconductor memories for data storage.

The main microcontroller components are [25]:

- Central Processing Unit (CPU) - consists of a control unit (executes instructions stored in the program memory), a logic unit, and an arithmetic unit (execute logical and arithmetical operations).
- Input and Output devices (I/O) - handle the data exchange with connected peripheral devices.
- Program Memory - Memory for storing the operating program (ROM, EPROM, etc.)
- Data Memory - Memory for reading and writing data during program execution (RAM)
- Bus System - System which connects all individual components of the microcontroller
- Clock Generator - Makes sure that all microcontroller operations are executed within a defined time window.
- Logic Circuits - Integrated into the individual I/O units and execute special tasks (e.g. Interrupts)

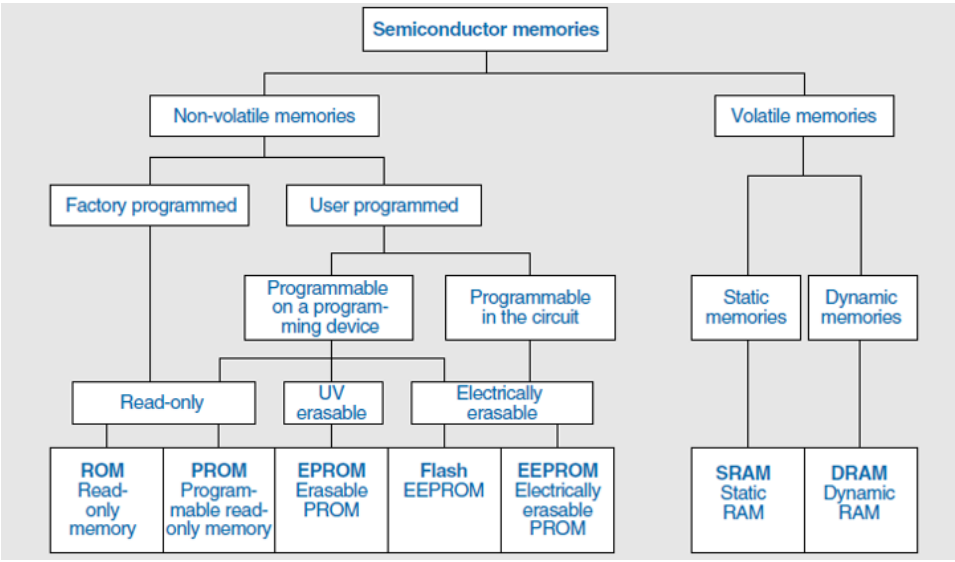

Figure 2.2: Semiconductor memories [25]

As already mentioned, memories are used for storing the program code and the data which is handled during the execution of a microcontroller application. They can be divided into two main parts: volatile and non-volatile memories. They differ in the sense that volatile memories lose stored data if the electric power is removed, while non-volatile memories retain stored information. Figure 2.2 contains a diagram of common memory forms that are combined with microcontrollers.

2.1.3 Embedded Automotive Software

The software of modern cars performs a wide range of tasks, starting with simple ones, such as seat height adjustment, to very complex tasks, such as predictive chassis or fuel consumption optimization. It is not unusual that software, the main innovator in the automotive industry today, contains up to 100 million lines of code in premium vehicles. [9] Approximately 60 percent of ECU development time is spent on creating the software. In order to provide the best quality, it is essential to use modern tools and processes. The most commonly used software development platforms in the automotive domain are Matlab/Simulink and SysML. The developed models are typically converted into C programming code. However, there is also a practical need for some software to be implemented directly in the C programming language. [9]

Embedded Automotive Development Process

The development process of embedded automotive software follows the so-called V-model. This model separates the concern and development steps, and thus allows explicit focus on a dedicated activity during the development process.

The development process is initiated with the analysis of the system requirements. That step is followed by the system design and software/hardware requirements analysis. It is common in this phase to implement a prototype software in order to test the concept before moving further in the development process. This step enables early error detection and thus uses resources

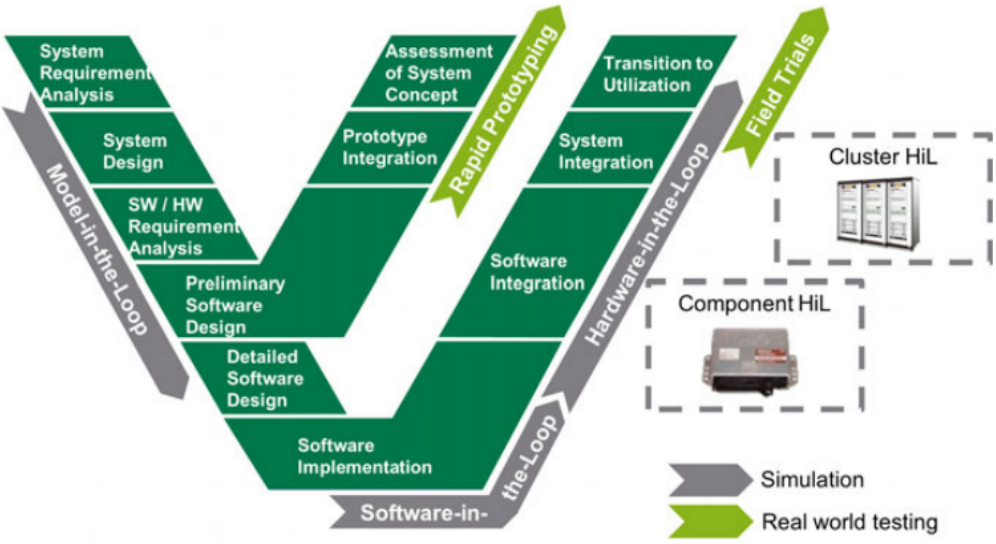

Figure 2.3: V-Model for automotive software development [23]

more effectively than it would be the case when allowing the propagation of errors and their discovery in the later development stages. For testing of these prototype implementations, Rapid Prototyping Systems (RPS) are used for fast integration of the prototype. When a prototype fulfills all the specified requirements, the development process proceeds with detailed software design (modules and functions) and finally to the software implementation. This concludes the verification of the system requirements. The next step is the validation of the implemented software. During validation, the software implementation is first tested without being deployed in the real hardware environment (Software-in-the-Loop). If the software performs its intended functionality, it is then integrated into real hardware and tested for faultless performance (Hardware-in-the-Loop). Any errors detected in these phases trigger a return to an earlier step of the V-model and modification of the software accordingly. The further steps are repeated, but with the improved components. Thus, the development process could involve many iterations trough the V-model, until all requirements are accomplished. Once the software's intended functionality is proved in the test hardware, it is then integrated into a real environment (f.e. a Vehicle) for final tests.

2.1.4 Rapid Prototyping

Since automotive subsystems are of critical nature, testing during the development of automotive subsystems has become a very important activity. Rapid prototyping is a very powerful method for identifying problems in the early stages of software development. It enables developers to test and verify new ideas and concepts efficiently before proceeding with the development process and implementation on the target ECU [14]. For this purpose, special rapid prototyping systems have been developed and are available on the market. They are equipped with very powerful hardware, thus having a few restrictions regarding computing power, memory storage, etc. This enables easy integration and testing of software functions, which are developed for much less powerful ECUs, without any hardware-specific considerations [26]. Furthermore, rapid prototyping systems are equipped with standard automotive communication interfaces such as CAN and FlexRay, providing them with various connection possibilities. By employing bespoke plug on devices (PODs), they are also able to connect to specific microcontroller families over the debug port, which enables direct memory access and very high transmission rates. If an RPS needs to access a signal which is not directly available in an ECU, it can acquire the necessary data by directly connecting to a sensor via appropriate interfaces [12]. However, the high performance and convenience that is provided by rapid prototyping systems are reflected in their high financial costs.

2.2 Communication Protocols

Automotive electronics are continually evolving in a search for the most appropriate solutions for the given time and user expectations. This made it possible to increase the complexity of the ECU functions. ECUs became more interrelated and dependant on information available from other ECUs in the system. To enable interaction between ECUs, the first solution was to use signal lines and exchange data between them. However, with rapid increase in the demand and volume of information that needs to be exchanged, this approach proved to be insufficient. In order to solve this problem, serial bus systems have been developed. They enable the exchange of large amounts

of data between all ECUs connected to the bus with a high transmission rate. The first bus system which was introduced to motor vehicles in mass production was the Controller Area Network (CAN) [25]. Since then, the CAN bus was established as the standard system in not only the automotive sector, but also in the majority of automation appliances.

Furthermore, as an alternative to CAN, the Local Interconnect Network (LIN) was developed in 1998. It specializes in data exchange between sensors and actuators. LIN has a simple protocol and simple sequence control, making it usable even in low-capability ECUs with no additional hardware required. This protocol is suitable for low data rates (up to 20 kBit/s) and has a limited number of possible participants (maximum is 16).[25]

Another commonly implemented protocol is FlexRay. It was developed in 1999 by BMW and DaimlerChrysler. Its purpose is to provide high transfer rates and a deterministic and fault-tolerant working manner while being easy to use and is simply expandable. During its evolution, a focus was set on its suitability for use in active safety systems. Hence, the domain where FlexRay is used the most is active safety systems, but also drivetrain systems. [25]

However, the development of more advanced functions in vehicles requires higher bandwidth than CAN, LIN, and FlexRay can offer. For this purpose, Ethernet has been identified as a very promising candidate for the automotive industry of the future. Automotive manufacturers tend to deploy Ethernet as a network backbone, but this approach is still in development. Today, Ethernet is used mostly for Diagnostics purposes, ECU flashing, Rear-view cameras, Infotainment, and Driver Assistance Systems. [17]

2.2.1 Controller Area Network CAN

During the development of CAN, one of the main goals was to eliminate the need for a central control element for communication. All devices should be able to connect to a bus and receive all information sent on the bus. A topology like this does not only provide favorable electrical properties, but it also ensures that the failure of one device in the network would not affect the functionality of the whole system. Furthermore, it enables easy network

extensions. New devices can be connected to the system with a little extra effort. [25]

CAN Nodes and Signals

The participants of a CAN network communication are referred to as network nodes. The nodes consist of three main components: a microcontroller, a CAN controller, and a CAN transceiver. The microcontroller is responsible for controlling the CAN controller, preparing data to be sent, and evaluating the received data.

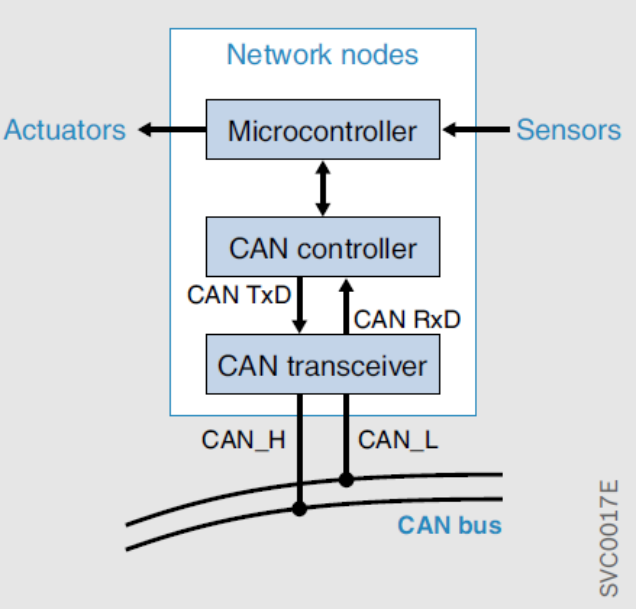

Figure 2.4: CAN node structue [25]

The CAN controller is responsible for managing the transmit and receive modes, while the transceiver takes care of providing the required voltage levels for transferring the messages over the bus.

Depending on the transmission speed, CAN networks can be divided into low-speed and high-speed buses. High-speed CAN uses transmission rates

from 125 kBit/s to 1 MBit/s while low-speed Can uses speeds from 5 kBit/s to 125 kBit/s.

In order to have a reflection-free communication, the communication lines must not have open ends. For this purpose, the bus lines are terminated at each end with a 120 Ohm resistor. Commonly, the terminating resistors are integrated into ECUs themselves.

Furthermore, the delay caused by the signal transit time across the lines also needs to be taken into account. This limits the maximum length of the signal lines for different transmission speeds.[25] The following recommendations are in place:

- 1 Mbits/s for 40 m
- 500 kBits/s up to 100m
- 250 kBits/s up to 250m
- 125 kBits/s up to 500m
- 40 kBits/s up to 1000m

CAN Protocol

In a CAN network, addressing does not occur by the individual network nodes, but by the sent messages. Each message has a unique identifier, which classifies the content of the message. Therefore, a node is able to broadcast a message to all other nodes (multicast). The receiving nodes read only those messages which are meant for them, so those whose identifiers are configured in their acceptance list. This ensures that every node decides for itself whether or not to accept a message sent on the bus. [25]

CAN Bus Access

Every node in a CAN network can try to send messages at any time. If this attempt is going to be successful or not, depends on the status of the bus. If the bus is in the ”recessive” state (it is unoccupied), each node is free to initiate the transmission of its messages. The message begins with a dominant bit (start-of-frame-bit), followed by the identifier. When

several nodes begin transmission at the same time, the system responds by employing ”wired-and” arbitration (arbiter = logical AND operator) in order to resolve the conflict. Every node sends the identifier of its message onto the bus one data bit at the time, with the most significant bit first. Each node that wants to send a message then compares the level present on the bus with the level it actually has. This way, a node that attempts to send a ”recessive” bit but encounters a dominant bit, will lose the arbitration process and stop the transmission. Thus, the message which has the lowest identifier also has the highest priority and is able to send the data first, without any data loss or delay (non-destructive protocol). The other nodes automatically become recipients of the message, and they try to resend their message as soon as the bus is free again. Without such access control, bus collisions would result in faults. To guarantee explicit bus arbitration, it is not allowed for more than one node to send messages with the same identifier. [25]

The consequence of such an arbitration process is that besides identifying the frame content, the identifier also has the role of prioritizing the frame during transmission.

Message format

The bit field of a standard CAN message frame is shown in figure 2.5.

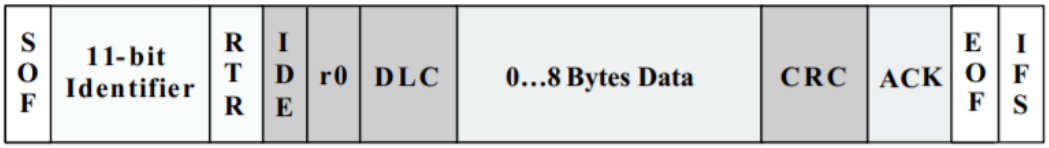

Figure 2.5: Bit field of a standard CAN frame[8]

The meaning of the bit fields are:

- SOF (Start Of Frame) - This bit marks the start of a message and is always a single dominant bit. It is used to synchronize all nodes on the network after being idle.
- Identifier - Unique identifier of a message and also establishes the priority of the message (lower identifier means higher priority).
- RTR (Remote Transmission Request) - If data is required from another node, this bit is set to be dominant. Even though all nodes receive this message, only the one specified in the identifier will respond.
- IDE (Identifier Extension) - set as a dominant bit in case a standard CAN identifier with no extension is being transmitted.
- ro - A reserved bit
- DLC (Data Length Code) - These four 4 contain information about the number of data bytes to be transmitted.
- Data - Here is the Data stored, and it can contain up to 64 bits.
- CRC (Cyclic Redundancy Check) - these 15 bits contain the checksum of the preceding application data for error detection.
- ACK (Acknowledgment) - If a node successfully receives a message, it overrides this recessive bit with a dominant one. In case that an error occurs and this bit does not get into the dominant state, the message gets discarded, and the sending will be repeated.
- EOF (End Of Frame) - This 7-bit long field marks the end of a message.
- IFS (Interframe Space) this 7-bit long field contains information about the time needed by the controller to move a successfully received message to its proper position in the message buffer.

2.2.2 Ethernet

The development of more advanced and complex software components in vehicles increased requirements for the communication protocols. As data volumes are increasing across the network, the demand for a communication protocol with a high bandwidth arose. CAN, LIN, and FlexRay were simply not designed to meet this requirement. As a good candidate for solving this problem, the automotive industry chose Ethernet. Ethernet is an open

LAN standard and defines the two lower layers of the OSI reference model (Physical and Data link). [17]

OSI model

The Open Systems Interchange (OSI) model provides a theoretical framework for a better understanding of data communications between two networked systems. The communication process is represented by seven layers that interact with each other [1]. The layers include:

Application Layer

The Application layer provides an interface for the end-user who is operating a device on the network. This is basically what the end-user sees.

Presentation Layer

The Presentation layer is responsible for how the data to be sent onto the network is formatted. It enables encryption and decryption of messages, compression and expansion, graphic formatting, etc.

Session Layer

The session layer provides services such as byte tracking, synchronization of data flow, acknowledgment of data received during a session, retransmission of data if it is not received by a device, etc.

Transport Layer

The Transport layer provides end-to-end communication between devices in a network. The most commonly used transport layers are TCP and UDP.

Network Layer

The Network layer provides an end-to-end logical addressing system, enabling routing of packets across several layer 2 networks. IP addresses are used in this layer for addressing the packets.

Data Link Layer

The Data link layer allows the device to access the local network (LAN) and send/receive messages. It provides a unique MAC address for packet addressing, which is physically connected with the hardware.

Physical Layer

The Physical layer is responsible for placing the bits on the media (wire, WIFI...) and also retrieving them. It defines the connector, interface specifications, and cable requirement.

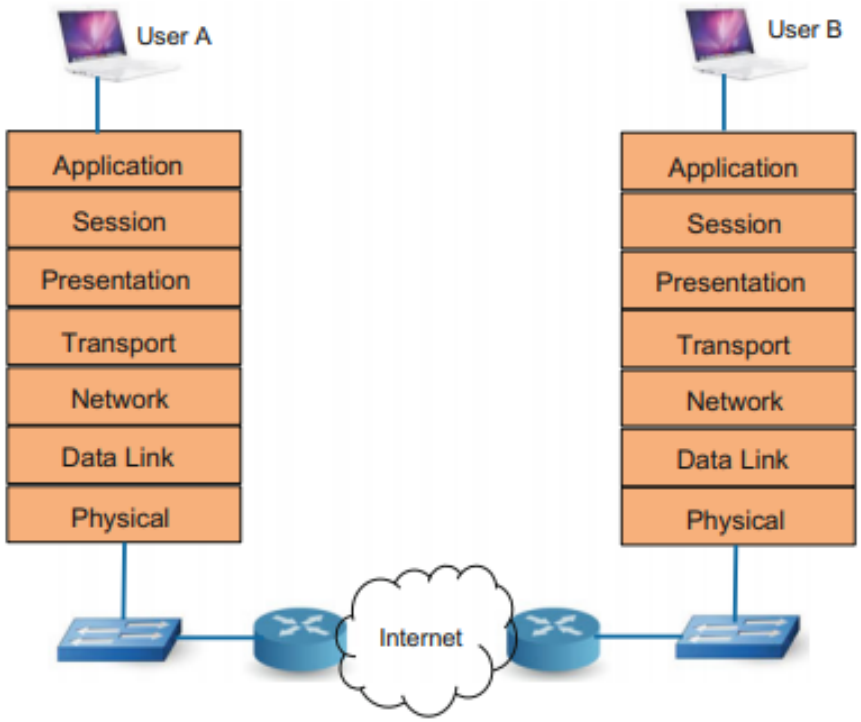

Figure 2.6: Ilustration of the OSI model[24]

When a device in a network receives a message, it first comes to the lowest layer (physical layer) in the form of an electrical signal and is then forwarded to all other layers in the chain. Thereby, the lower layers are more hardware-oriented while the upper ones are more software-oriented [19]. In case of

sending a message, it travels all the way down from the application layer to the physical layer, where it is going to be prepared for sending over a media. During the travel of the message through all the layers, every layer adds its own information (headers) to the front of the data it receives from a layer above it [24]. This process is called encapsulation.

Even though the network and transport layer are theoretically separate, in practice, they are very closely linked to each other. The well-know protocol used for the internet called Transmission Control Protocol/Internet Protocol (TCP/IP) comes from the transport layer protocol (TCP) and the network layer protocol (IP) [24].

TCP/IP and UDP/IP

Transmission Control Protocol (TCP) is one of the two most used transport protocols for communication over Ethernet. It is a connection-oriented protocol, which supports a packet retransmission mechanism and message acknowledgment. In the case of lost packets, TCP retransmits the data until either a connection timeout has occurred or a successful delivery [24]. This however, can cause high transmission latency. For this thesis, UDP/IP has been used and explained in more detail.

User Datagram Protocol (UDP) is another protocol that provides transport services to applications. Unlike TCP, this transport protocol does not have flow control or acknowledgment of received data. This however, ensures minimal overhead. Messages which do not pass the checksum test are simply dropped. Besides the logical addressing with IP, UDP provides additional addressing in the form of port numbers. Using port numbers, multiple applications and services are able to set up a connection on the same end. [22]

A UDP frame is also called a UDP datagram. It consists of an 8-byte header and a Data field.

The header consists of a source port number field, destination port number, total length, and checksum. Each of those fields is 16 bits long. Additionally to the UDP datagram, an IP pseudo-header is appended.

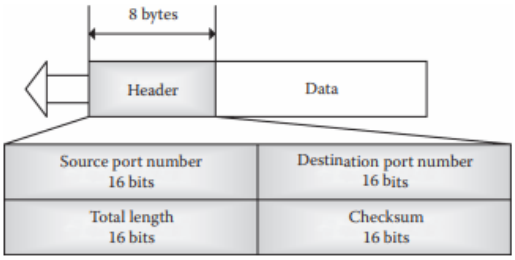

Figure 2.7: UDP Datagram format[22]

It contains five fields: source IP address, destination IP address, zeros, protocol field, and UDP length.

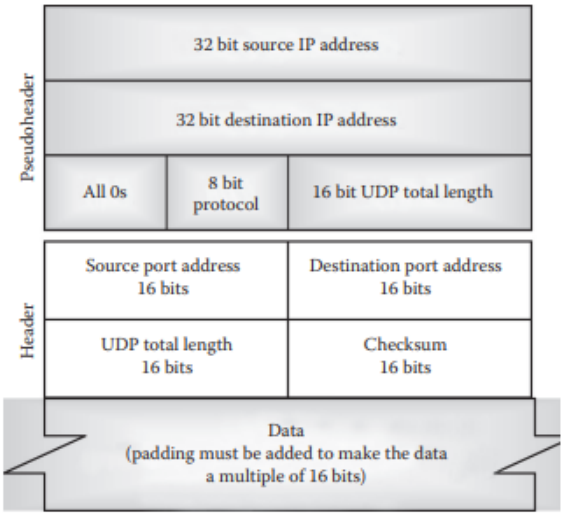

Figure 2.8: Pseudoheader with the UDP datagram[22]

2.3 Calibration of Electronic Control Units

2.3.1 CAN Calibration Protocol - CCP

The CAN Calibration Protocol (CCP) is the predecessor of the Universal Calibration Protocol (XCP) and was developed and introduced in 1992 by the manufacturer of calibration systems Ingineurbüro Helmut Kleinknecht. It utilizes the Controller Area Network (CAN) for communication and is a very often used calibration protocol in the automotive industry. CCP was taken over by the Standardization of Application/Calibration Systems task force (ASAP) in 1995. The conceptual idea of the CAN Calibration Protocol was to permit read and write access to internal parameters of an ECU over the Controller Area Network (CAN). The working principle of CCP is the same as XCP on CAN, and more information can be found in section XCP on CAN.[3]

2.3.2 Universal Calibration Protocol - XCP

The Universal Calibration Protocol (XCP) is an ASAM standard, which was first introduced in 2003 and is mainly based on the experience gained from working with the CAN Calibration protocol. ASAM stands for ”Association for Standardisation of Automation and Measuring Systems” and represents a working group that includes vehicle OEMs, tool manufacturers, and suppliers. This Standard defines a generalized measurement and calibration protocol, which is independent of the specific transport medium, unlike CCP, which supported only CAN as the transport layer. This was achieved by dividing XCP into a Protocol layer and a Transport layer. Depending on the Transport layer, one can refer to XCP on CAN, XCP on Ethernet, etc. The ”X” in XCP stands for the variable and interchangeable transport layer.[6]

2.3.3 ECU Interfaces

Another method for measurement and calibration of ECUs, which is often used, is utilizing a physical ECU interface in the form of a Plug On Device

(POD). By connecting this POD to the debug port of the ECU, it provides direct access to its memory for calibration tools [16] [11]. Companies that offer rapid prototyping and calibration tools also develop PODs for their products. However, PODs are not universal for all microcontroller families, thus depending on the microcontroller of the ECU, a different POD has to be considered [16] [11]. Some of the manufacturers of such products are ETAS, dSpace, and Vector. From stand-alone rapid prototyping units such as ES910 from Etas or MicroAutoBox from dSpace to measurement and calibration interfaces such as VX1000 from Vector can all be found on the market. The usage of PODs enables those devices data transmission with very high speeds. The obtained data is then processed within the device itself, or forwarded to another calibration tool [15][12][28]. Devices used as measurement and calibration interfaces provide the data they obtain from ECUs over an XCP on Ethernet interface to another calibration tool (CAPane, INCA, etc.). They rely on other calibration tools and cannot work independently[28]. RPS devices however, are able to work in stand-alone mode and do not need to be connected to another calibration tool, but they also have this option available [15][12]. For applications where high data throughput is not needed, those devices also provide standard interfaces such as CAN and FlexRay for connecting to ECUs and performing measurement and calibration operations with XCP.

2.4 XCP Protocol

2.4.1 XCP Protocol Layer

The mechanism which enables measurement and calibration of ECUs is defined in the Protocol layer of XCP. Since XCP is based on the master-slave principle, data between a master and a slave is exchanged in a message-oriented way. The whole XCP message (XCP frame) is embedded into the data field of the underlying transport layer (in case of Ethernet, for example, in a UDP packet). As with many communication protocols, the XCP frame consists of three parts: XCP header, XCP packet, and XCP tail, as shown in the figure 2.9.

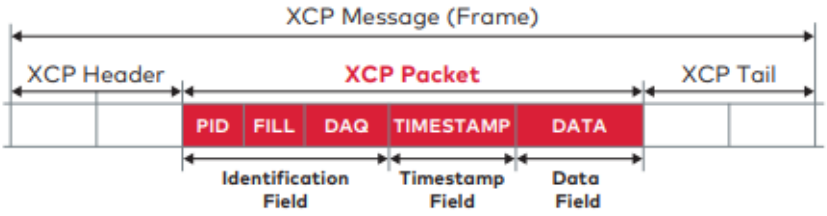

Figure 2.9: XCP frame [2]

The XCP header and XCP tail depend on the used transport layer, while the XCP packet is independent of it. The XCP packet also consists of three components: Identification Field, Timestamp Field, and Data Field.

The Identification field always starts with the Packed Identifier (PID), which identifies the packet and ensures that both the master and slave are able to determine the meaning of the message. Based on the data exchange method (Command Transfer Object (CTO) or Data transfer Object (DTO), explained later), the structure of the Identification Field can vary. In the case of CTOs, it is fully sufficient to use only the PID in the Identification Field to identify the message. In the case of DTOs, the Identification Field can also contain other elements (FILL and DAQ), which will be described in a later chapter. Furthermore, DTO packets can also utilize the optional timestamp fields for providing the time information alongside the measurement values. The Data Field of the XCP packet is used to provide the measurement data from the slave, but also specific parameters required for the different commands in case of CTOs.[2]

2.4.2 XCP Transport Layer

XCP protocol was designed to support many different transport layers. As of today, the following layers have been defined: XCP on CAN, FlexRay, Ethernet, SxI, and USB [6]. This chapter will focus on XCP on CAN and XCP on Ethernet since these transport layers have been used in the implementation. In order to find implementation descriptions of other transport

layers, one can refer to the website of ASAM and download the Standard documents[1].

XCP on CAN

Since XCP is the successor to CCP, it is natural that it satisfies the requirements for the CAN bus. As multiple CAN nodes can be connected to a network, CAN identifiers are used for controlled transmission of messages between them.

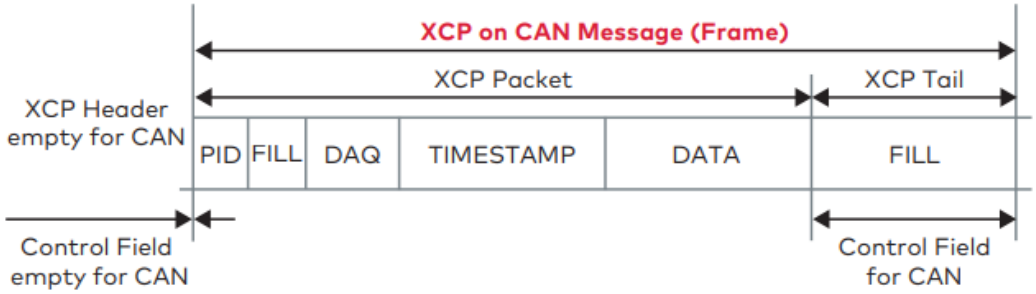

Figure 2.10: XCP on CAN message format [2]

The XCP communication uses at least two CAN identifiers for each unique slave [4]. One CAN identifier is used for transferring XCP frames from master to slave (CMD, STIM), and the other identifier is used for transferring XCP frames from slave to master (RES, ERR, EV, SERV, DAQ). The CAN identifiers used for XCP communication must not be used for any other purpose on the CAN bus. The configuration of CAN messages and XCP objects is normally defined in the slave description file (A2L), which is then used for configuration of the master. The messages sent from master to slave should be configured to have a higher priority than the response from the slave. XCP on CAN supports the standard and block transfer communication model, while interleaved communication is not allowed.[2]

Since XCP frames are encapsulated into the data field of the transport layer, it is limited to 8 bytes in the case of CAN. The first byte of the XCP frame is

[1]https://www.asam.net/standards/detail/mcd-1-xcp

always used for the identification of XCP commands, thus only seven useful bytes are available for transporting useful data.

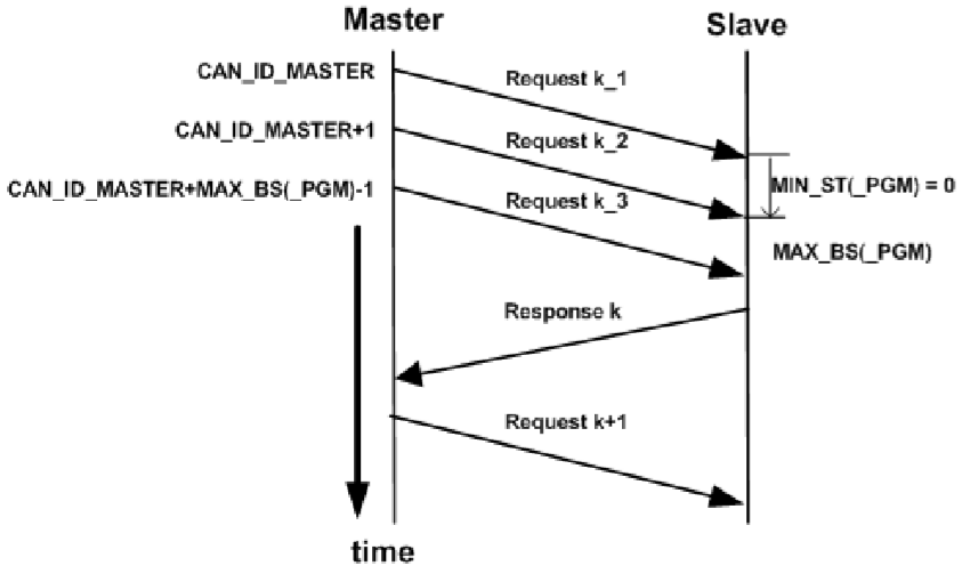

Figure 2.11: XCP on CAN block transfer [2]

XCP on Ethernet

XCP on Ethernet can either be used with TCP/IP or UDP/IP. The main difference is the ability to detect packet losses. TCP/IP uses a handshake method and organizes repetitions in case of a packet loss, while UDP/IP does not have this mechanism, and packet losses are being ignored. UDP/IP has, therefore, a minimal overhead and is suited for sending measurement data, where a measurement gap caused by a packet loss is not that critical [2]. In case that the measurement data is to be used as the basis for fast control, TCP/IP is recommended. XCP on Ethernet uses the standard communication model, while block transfer and interleaved are optional [5].

Figure 2.12 shows the message format of XCP on Ethernet.

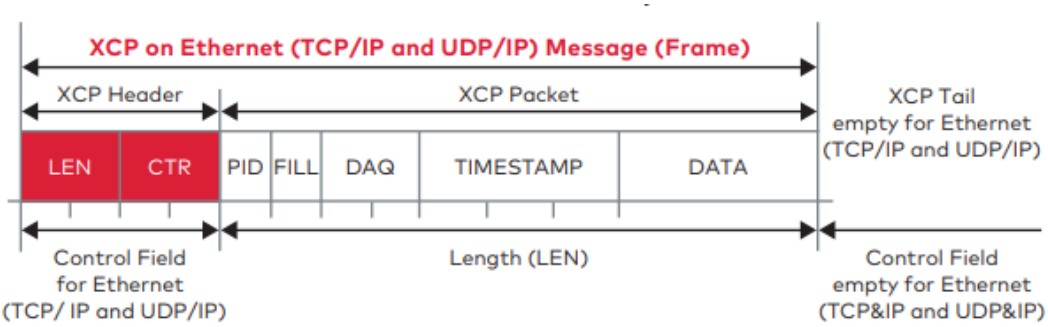

Figure 2.12: XCP on Ethernet [2]

As with all transport layers, the XCP frame is embedded into the data field. The header fo the XCP frame consists of a LEN and CTR field. LEN contains information about the number of bytes in the XCP packet, while CTR is used as a counter for detecting packet losses. The CTR value gets incremented every time a message is sent. The master and slave increment their counter independent of each other. This way, packet losses are easily detected.[2]

The XCP packet comes after the XCP header. With XCP on Ethernet, there is also no tail. The maximum size of the XCP packet is again limited by the size of the UDP/IP packet.

2.4.3 Command Transfer Objects - CTO

CTOs are used for transferring commands from the master to the slave, but also the Response from the slave to the master. Each time the slave receives a command from a master, it has to provide either a positive or a negative Response. The first Byte of the Response in hexadecimal form is **0xFF** in case of a positive response, and **0xFE** in case of a negative response. Other Bytes of the positive or negative Response are used for specific parameters that provide some additional information. For example, when a master sends the command to connect to a slave and it sends back a positive response, alongside the positive response, it also provides additional information about which features it supports, maximum packet length, etc. The full details can be found in the ASAM Standard. During an XCP session, the

exchange of commands and reactions between master and slave can happen in three different modes: Standard, Block, and Interleaved.[2]

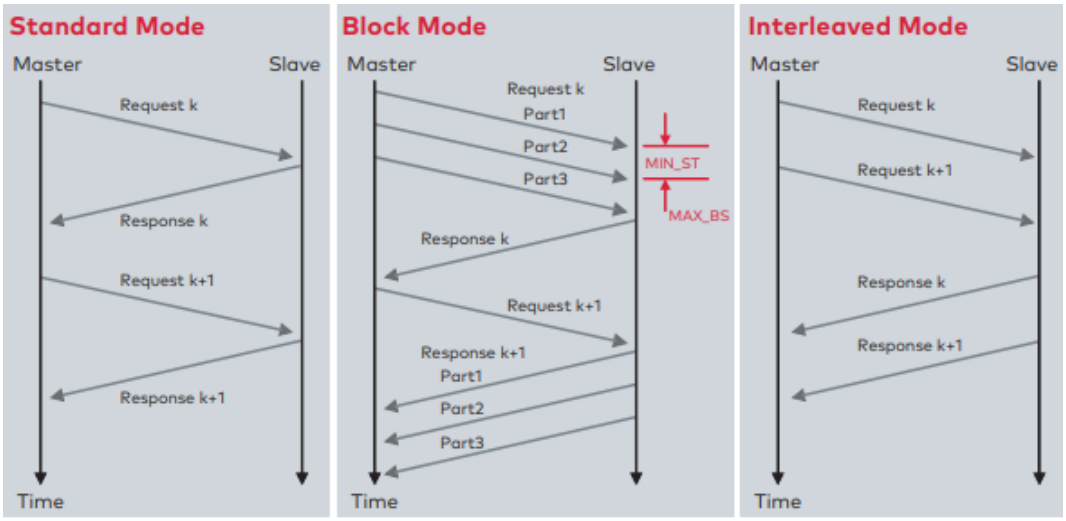

Figure 2.13: XCP protocol modes [2]

In the standard mode, after each request from a master comes only a single response from a slave. Hence, transmitting a large amount of data (DOWNLOAD, UPLOAD, FLASHING) with the standard mode would take a lot of time. To speed up the transmission of a large amount of data, the Block mode can be used [2]. It enables sending multiple messages in one direction at once. However, this mode is heavily reliant on system performance. It is necessary to limit the total number of commands which can be sent (MAX_BS) and to maintain the minimum times between two commands (MIN_ST). Another mode that enables faster communication than standard mode is the interleaved mode. In this mode, requests can be sent consecutive, without waiting for a response. The slave stores the requests and sends the corresponding responses. However, the slave can store only a limited amount of requests, and the master has to take care that this limit is not crossed. [2] A master can send the GET_COMM_MODE_INFO command to obtain information about which mode is supported by a slave [6].

2.4.4 Data Transfer Objects - DTO

DTOs are used for synchronous data transfer between a master and a slave. For synchronous data transfer, two different modes are available with XCP. One mode is for synchronous data transfer from a slave to a master (DAQ mode), and the other mode is for synchronous data transfer from a master to a slave (STIM mode). For using any of those modes, the first step is configuring the data which needs to be synchronously measured/calibrated. After the configuration phase, the master sends the command for the start of data transfer. Hereby, the slave uses internal events for triggering the data transfer.[2]

2.4.5 Data Polling with XCP

Polling is the simplest measurement method that is solely based on CTOs [2]. The XCP-Master uses a CTO command (SHORT_UPLOAD) to request a parameter value from an XCP-Slave, and the XCP-Slave responds with a CTO which contains the value of interest. The simplicity of this method brings drawbacks which limit its practical usefulness. The XCP specification defines that every measurement parameter has to be polled individually. This means that for every single measurement parameter, two messages need to travel over the bus. One message is the request from the master, and the other messages are the Response from the slave. This causes the busload to be increased. Furthermore, another drawback of this method shows itself when multiple measurement parameters are being polled. The tendency is usually that all measurement parameters correlate to each other. However, since every single measurement parameter has to be polled sequentially, the measurement values will not necessarily be in correlation with each other. The time needed for sending all poll requests is much higher than the computational cycle of the ECU. Hence, the measurement values will all be from different computational cycles. From this, the two main drawbacks of this method can be seen [2]:

- Unnecessarily high bus traffic
- Measurement values are not evaluated in relation to the process flows in the ECU

To overcome these drawbacks, the Synchronous Data Acquisition (DAQ) measurement method is used.

2.4.6 Synchronous Data Acquisition with XCP - DAQ

Synchronous Data Acquisition DAQ solves the problems present in data acquisition with polling. The most significant advantage with DAQ is that the busload is much more decreased, as no requests from the master have to be sent after the configuration phase. The bus is loaded just by the DAQ data from the slave. Furthermore, since the DAQ data from a slave is linked to an internal slave event, the correlation of the measured values is also achieved [2].

For example, if an event is set to be triggered at the end of the computation cycle, each time the slave reaches the "Computational cycle completed" event, it will gather all the measurement parameters, save them in a buffer and send them to the master. These XCP events in the slave can be cyclic as in the example before but do not have to be. For instance, in the case of an engine controller, an XCP event can be dependant on the rpm of the engine (angle-synchronous) and therefore be asynchronous.[2]

For starting a DAQ measurement, it first needs to be configured. The configuration is performed by the master by sending a defined message sequence to the slave. During the configuration, the master tells the slave which signals should be measured, how to distribute the measurement data into messages the master can interpret, and to which event the measurement data should be linked. For describing the sequence in which the slave should assemble the bytes into messages, Object Description Tables (ODTs) are defined. They contain addresses and object lengths of all parameters which need to be measured, assembled into a message on the bus. This message is transmitted as a DAQ DTO (Data Transfer Object) [2].

After the configuration is completed successfully, the slave waits for the master to send the "START MEASUREMENT" command. After the master sends this command, it starts to listen to the bus. The slave then begins to send the DAQ data when an event occurs. Since the master defined the allocations of all individual objects itself, it can interpret the individual data

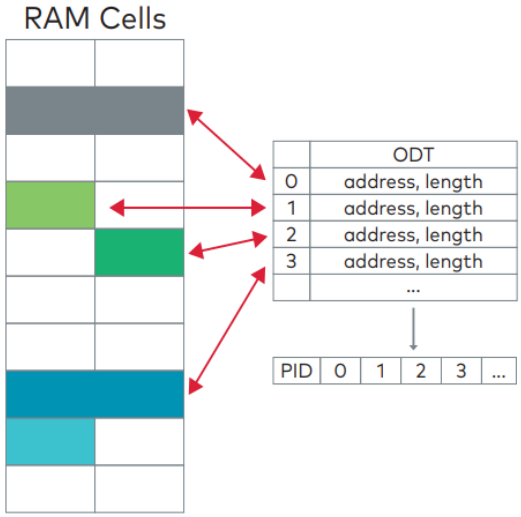

Figure 2.14: Allocation of RAM addresses to DAQ DTO [2]

it receives from the slave. The slave stops the transfer when the master sends the "STOP MEASUREMENT" command. The number of entries an ODT can have is limited by the used transport medium. For example, in the case of CAN, the maximum number of useful bytes is seven.[2]

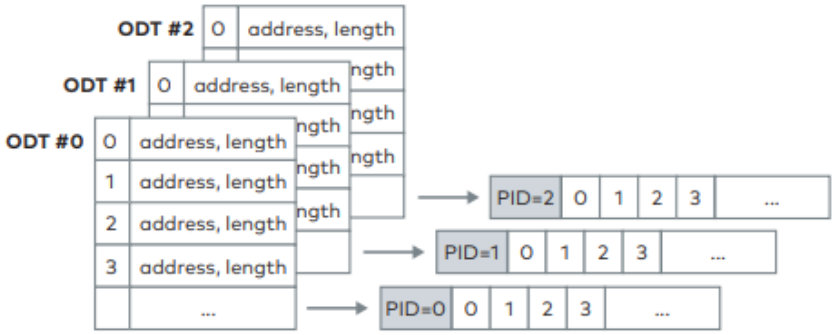

Figure 2.15: DAQ lists with three ODTs [2]

When more data needs to be sent in this case, one ODT is not sufficient, and more have to be defined. The slave must then be able to copy the data into the correct ODT and the master to correctly identify the received ODTs. The ODTs are then combined into DAQ lists, which are assigned to an

XCP-event. This means that one DAQ list can contain only the ODTs which are assigned to the same event.[2]

If, for example, two ODTs are defined for measuring data in two different measurement intervals (two separate events), then also two DAQ lists have to be defined. Per event used, one DAQ list is needed. Furthermore, DAQ lists can be defined as static or dynamic.

Static DAQ lists

Static DAQ lists are permanently defined within the ECU code and cannot be changed by the user [6]. The content of the ODTs and DAQ lists is still configurable, but the framework that can be filled is unchangeable. If an ECU has multiple static DAQ lists defined and an attempt is made to measure more signals with an event than it fits a DAQ list, it will be terminated with an error. It does not matter that other DAQ lists are not even used and available. In contrast to this, there are also Predefined DAQ lists. They have not only the framework but also the content of all ODT entries defined. This method is rarely used in practice since it does not provide any flexibility to the user[2].

Dynamic DAQ lists

Dynamic DAQ lists are a special aspect of the XCP protocol. In this case, DAQ lists and ODTs are not permanently defined in the ECU code, but only parameters of the memory area that can be used for the DAQ lists [6]. The XCP-Master has the freedom to define the DAQ lists and ODTs in an XCP-Slave ECU according to its needs. There are various functions defined in the XCP standard, which can be used by an XCP-Master for configuring dynamic DAQ lists.[2]

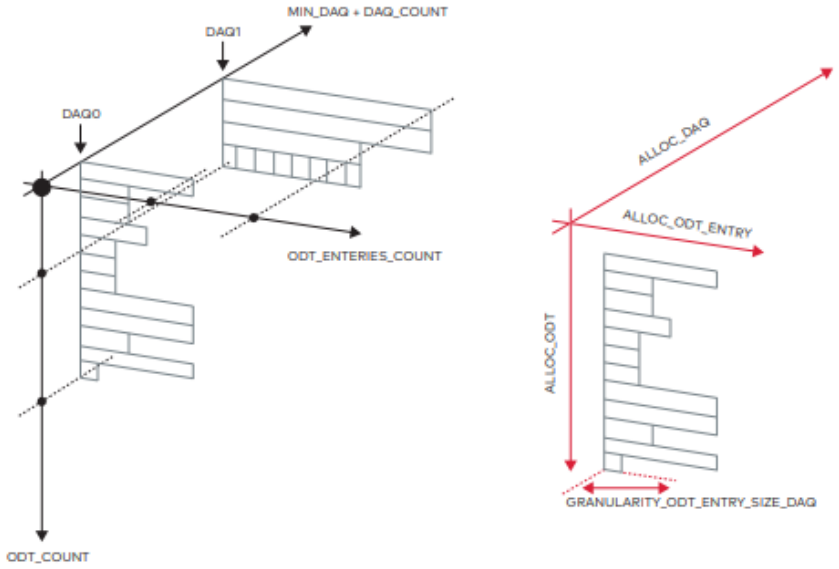

Figure 2.16: Dynamic DAQ lists [2]

2.4.7 Calibration with XCP

The process of changing the value of a parameter in an XCP-Slave is called calibration. Hence, in order to calibrate a parameter in an XCP-Slave, the XCP-Master must send the parameters memory location address and the new value itself to the slave [2]. The addresses in XCP are always defined with five bytes. Four of those bytes are used for the actual address and the remaining byte for the address extension. When CAN is used as the transportation medium, there are only 8 bytes at disposal for encapsulating XCP messages. One byte is reserved for the PID, which makes only seven useful bytes available. If one would try to calibrate a 4-byte value, it would need nine useful bytes to transfer this message. Therefore, the calibration is performed with two commands from the master: SET_MTA and DOWNLOAD. The first message to be sent is SET_MTA from the master. SET_MTA is used for sending the address to which a new value should be written. Afterward, the slave responds with an acknowledgment or error message. The third message in the chain is the command DOWNLOAD from the

master. DOWNLOAD specifies the length of the value to be written (number of bytes), as well as the value itself (HEX format). The slave must again send a positive or negative response, which concludes the calibration process. As the calibration process changes the value of a parameter in the ECUs RAM, the new value will be used by the application. However, if the ECU should reboot, the calibrated value in RAM would again be overwritten with the original value from the flash. To ensure that the calibrated value is stored permanently, there are two possibilities[2]:

- The new parameter could be saved in the ECUs EEPROM automatically when ramping down the ECU or manually by the user. In order for this to work, it must be ensured that data can be saved in a non-volatile memory of the slave (EEPROM or Flash). This method is however, rare, since ECUs usually do not have memory space to spare.
- The parameters could be saved in the form of a file on a PC. This method is the preferred one. The calibration values can be saved in different file formats, with the simplest to be an ASCII text file.

Furthermore, calibrated parameters can also be flashed into the ECU. This process will store the values permanently into the flash memory of the ECU. The easiest way would be to transfer the set file, which contains the new calibrated values into a C or H file, and to perform a compiler/link run to generate a new flash file. However, this method requires access to the source code of the ECU, which is often not available. Furthermore, it could also take a considerable amount of time to perform this, depending on the parameters of the code. If, alternatively, the original Flash file of the ECU is available, it could be modified with the new calibration set file. The possibility to do this is provided by CANape. CANape takes the addresses and values from the parameter set file and updates those values in the flash file, generating a new modified flash file with updated values.[2]

2.4.8 Stimulation with XCP

Stimulation with XCP works on the same mechanism as DAQ, but now in the opposite direction. While DAQ mode is used for synchronous transmission of measurement data from slave to master, Stimulation is used for

synchronous transmission of calibration data from master to slave. Communication during Stimulation is therefore synchronized to an event in the slave. It is necessary for the master to know, to which events in the slave it can synchronize at all. For the slave, it is important to know the location in the packets at which the calibration parameters can be found. This information must be contained in the A2L file [2].

2.4.9 Flashing with XCP

Flashing is a process of updating the software that runs in an ECU. This is done by changing the data in the area of flash memory. For successful flashing, it is vital to know how the memory of the ECU is laid out. Flash memory consists of multiple sectors, which are described by a start address and a length [2]. In order to distinguish them, every sector gets a consecutive identification number. As for this number only one byte is available, there can be a maximum of 255 sectors. Before the process of flashing can begin, an executable code, which is referred to as flash-kernel, is sent to the slaves RAM. This code handles the communication with the XCP master during the flashing process, as well as erasing the flash memory before it can begin. The whole flashing process with XCP is subdivided in three phases:

- Preparation
- Execution
- Post-processing

During the preparation phase, things such as version control (checking if new contents can even be flashed) can be performed. During the execution phase, the new contents are sent to the ECU. During the Post-processing phase, things such as checksum checking are performed.[2]

2.5 Intelligent Watchdogs

All computer systems are prone to errors. Embedded systems are no exception. Faults in a system can be caused by various factors, from radiation influence coming from the environment, to bit flips randomly happening

during the writing of data into RAM. While some faults are only temporary, others can cause the system to fail permanently. In order to detect such faults, system monitors and general fault detection schemes such as watchdogs are being used. They are systems much less complex than the system they are monitoring and can be implemented on the board of the monitored system itself, or connected to it as an external device. After detecting a fault, watchdogs take action and try to restore the system to its former, fully functional state. [20]

In order to monitor a system, watchdogs require input data about the system states. This data is evaluated by the watchdog, and in case of unexpected behavior of the system, the watchdog acts accordingly. Those actions could be setting an alert, trigger some signals, but also resetting the whole system.

When watchdogs perform more complex actions than just triggering a signal or executing a command, we talk about intelligent watchdogs. They usually have more advanced algorithms for evaluating the system states and a more complex approach of deciding what action to take. In the automotive industry, watchdogs are often used in safety-critical applications where faults can not be tolerated at all.

One area where fail-operational behavior is essential and where a lot of focus is set today is automated driving. For users to accept autonomous vehicles, it has to be trustworthy, safe, and secure. It must be capable of independently handling safety-critical situations [21]. To ensure this, the safety-critical components of a vehicle must have redundant systems and a watchdog, which will monitor everything and take actions if any faults should occur.

For gathering data from an ECU that needs to be monitored, Watchdogs could utilize the XCP protocol and thus have direct access to its memory.

3 Design and Implementation

3.1 Concept

This work is based on the utilization of the XCP protocol and the implementation of an XCP-Master Controller for the chosen platform. As a result, the platform has three aspects of its usage: its implementation aids communication with other XCP-Slave devices as a rapid prototyping system, it acts as an intelligent watchdog, and also provides an Ethernet-CAN interface for situations where an adapter is not available.

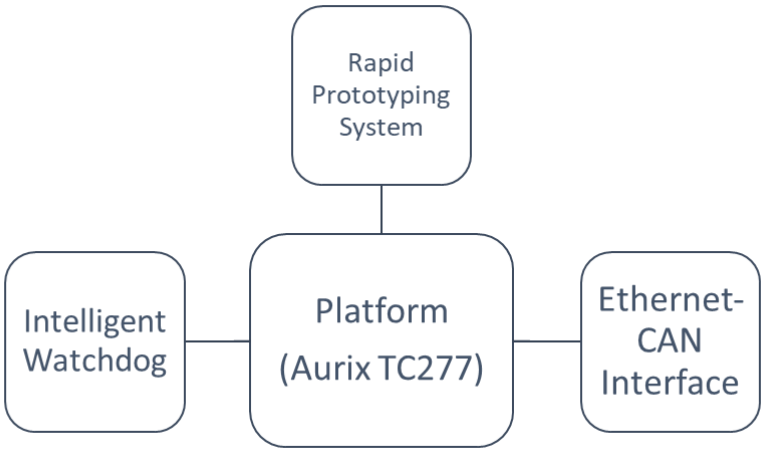

Figure 3.1: Proposed platform usability

For having RPS functionalities, the platform has to utilize an XCP-Slave driver and an XCP-Master Controller. The purpose of the XCP-Slave driver is to provide access to other calibration tools such as CANape and enable

configuration of the RPS parameters during run-time. The XCP-Master Controller provides functions for connecting to an XCP-Slave and perform measurements and calibration, and with the combination of those two, bypass of functions. This bypass functionality enables the testing of functional changes without having to perform a flash update directly on the target ECU. Instead, it is flashed to the RPS and used to bypass the corresponding function in the target ECU.

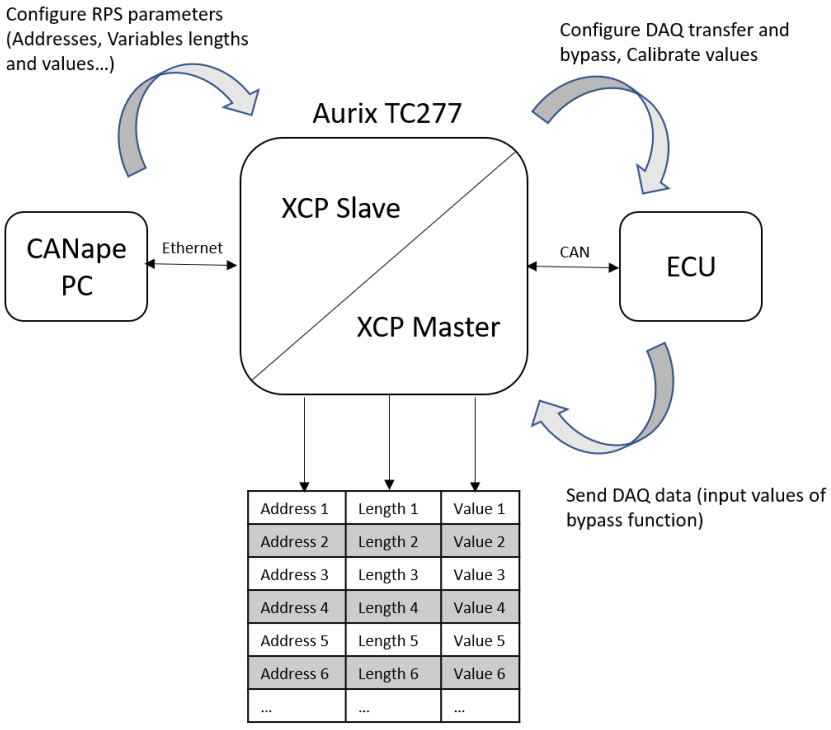

Figure 3.2: RPS concept

For storing information about the signals and parameters to be measured and calibrated in target ECU, two buffers with the same structure are defined in the RPS. They contain information about the memory address of the parameter, size in bytes, and the value of the parameter which is measured or going to be sent for calibration.

One buffer is used for storing measurement-related data, one for storing calibration-related data, including calibration-enabling switch variables of

the target ECU. A more detailed explanation is given in the RPS section. Figure 3.2 shows the concept of the RPS functionality.

The XCP-Master Controller can also be utilized as an intelligent watchdog for monitoring other ECUs' behavior. By having direct access to its memory, it is able to obtain and evaluate any required data. Safety-critical data must be faultless during the whole operation of the ECU. The monitoring of such data is crucial for guaranteeing system dependability.

The intelligent watchdog can ensure that the values are in allowed boundaries, and if violations occur, it could act by sending proper commands. Furthermore, an Intelligent watchdog could be configured to execute the same functions as the target ECU and with the same input values. By comparing the output values of the target ECU and the values it calculated itself, it can ensure the consistency of the data.

Figure 3.3 shows the concept of intelligent watchdog functionality.

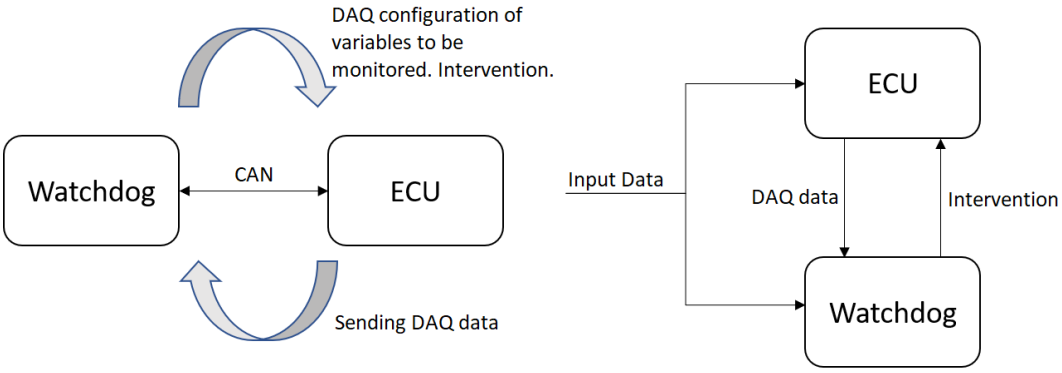

Figure 3.3: Intelligent Watchdog concept

The utilization of UDP/IP and CAN transport layers sets the foundation for the implementation of an Ethernet-CAN interface. With proper message handling, the payload of UDP/IP messages can easily be extracted, formatted, and forwarded to the CAN bus. Figure 3.4 shows the concept of the Ethernet-to-CAN interface.

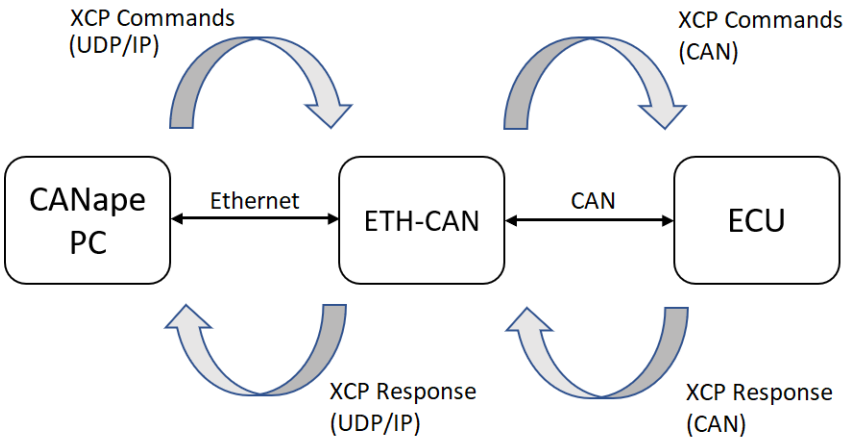

Figure 3.4: Ethernet-to-CAN interface concept

3.2 Implementation

3.2.1 Toolchain Selection

Tools used during the development phase of this project are listed and described in table 3.1.

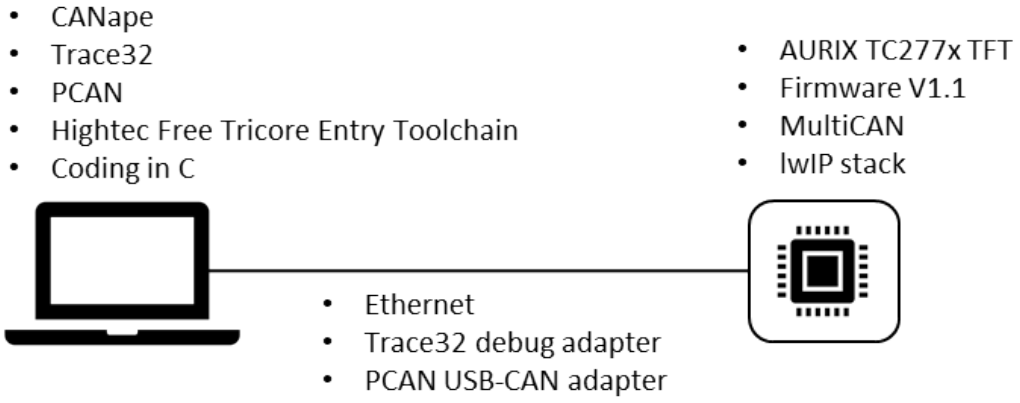

Figure 3.5: Toolchain during development

item	description
Aurix TC277 evaluation kit	Chosen as the platform for the implementation of this work. This evaluation kit if fully equipped for demonstration of all relevant hardware components.
”AURIX TC277x TFT Application Kit Firmware V1.1”	the basis for the software development.
Hightec Free Tricore Entry Toolchain	IDE for developing software in the C programming language
CANape	Calibration tool from Vector GmbH utilized for performing XCP communication with the platform as an XCP-Slave
Trace32 with adapter	Debugging tool from Lauterbach used for debugging during development
PCAN with adapter	CAN communication tool from Peak Systems used for tracing and debugging CAN messages
MultiCAN	Library from Infineon for utilizing the CAN module on the board
lwIP	Open-source Ethernet stack for utilizing the Ethernet adapter on the board
Spyder 4.1.1	IDE for developing the configuration tool using Python 3.7

Table 3.1: Toolchain Description

Figure 3.5 shows which tools are used on the side of the development PC and which tools are deployed and running directly on the target platform. Besides Ethernet for communication between those two, adapters from PCAN and Trace32 have also been used for debugging purposes.

3.2.2 Software Architecture

Figure 3.6 shows the overview of the software architecture, indicating the implemented and integrated software modules and how they are linked.

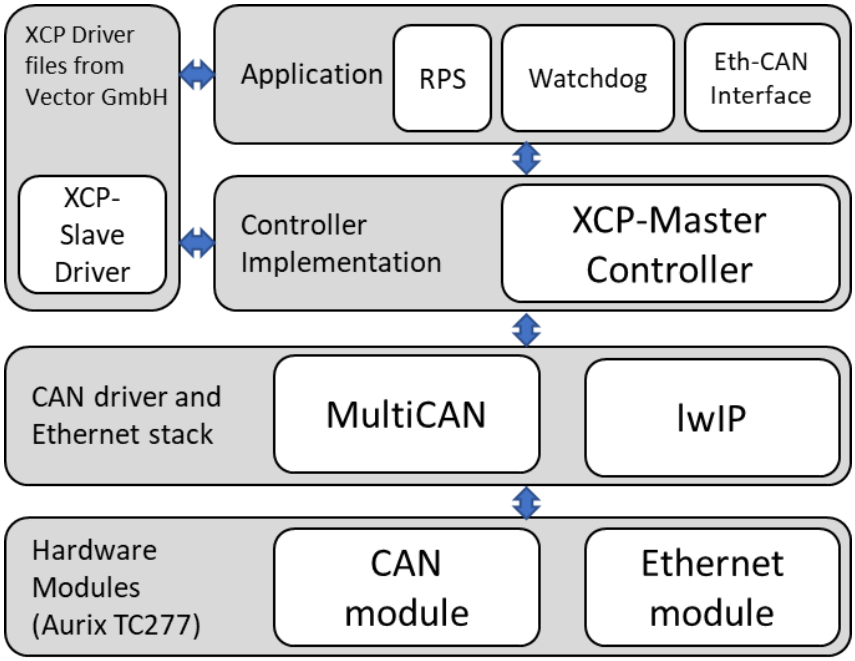

Figure 3.6: Software Architecture Overview

The application includes three different working modes: RPS, watchdog, and Ethernet-CAN interface. The application layer is communicating with the XCP-Master Controller, which is responsible for coordinating the process and CAN/Ethernet message flow. Furthermore, the XCP-Master Controller uses the MultiCAN and lwIP drivers for receiving and transmitting messages over the CAN and Ethernet bus, respectively. The XCP-Slave driver is linked to the XCP-Master Controller, which provides access to the transport layers, and to the Application layer. This ensures that the XCP commands for application configuration, which may come from another XCP-Master device, can be received, interpreted, and applied.

3.2.3 XCP-Slave Implementation

In order to have an online-configurable RPS, it has to contain an XCP-Slave driver implementation. Vector GmbH provides a basic version of this driver as free-to-use software with limited functionalities compared to the professional version, but enough for this work. All mandatory XCP Commands are available. The source files containing the XCP protocol layer implementation can be downloaded from the official Vector GmbH website. However, no transport layer implementation is provided. Therefore, a transport layer implementation is needed in order to integrate the XCP driver into the project. The configuration tool for the RPS is going to be CANape, so that the transport layer for the XCP-Slave implementations has been chosen to be Ethernet, with UDP/IP as the underlying transport layer. This is because of the following reasons:

- Both the ECU and the PC have an Ethernet interface and do not require any additional hardware
- Most calibration tools use XCP over Ethernet by default [12][15]
- It is easily configurable

UDP/IP has been chosen since it provides minimal overhead, and the mechanisms of TCP/IP are not really needed for this project. An Ethernet transport layer implementation already exists in a project, which is available on Github[1]. This implementation was integrated into the master thesis project with additional adaptions and configurations.

Communication

As XCP supports a variety of transport layer protocols, the XCP protocol layer provides three generic API calls for interaction between them. Those API calls are:

- XCPCommand
- XCPSendCallback
- ApplXCPSend

[1] https://github.com/shreaker/OpenXCP

The ***XCPCommand*** function is called within the transport layer when data has been received from a master. This function forwards the XCP payload to the protocol layer, so that it can be interpreted and evaluated. The ***XCPSend-Callback*** function is used by the transport layer to inform the protocol layer about a successful transmission of an XCP packet. The ***ApplXCPSend*** function is called by the protocol layer for transmitting an XCP packet to the transport layer. After this, the transport layer needs to embed this XCP packet into the specific frame which it uses.

Furthermore, there are also API calls for the interaction of the protocol and the application layer. Figure 3.7 illustrates the principle and API calls used for communication between the protocol layer, transport layer, and the application.

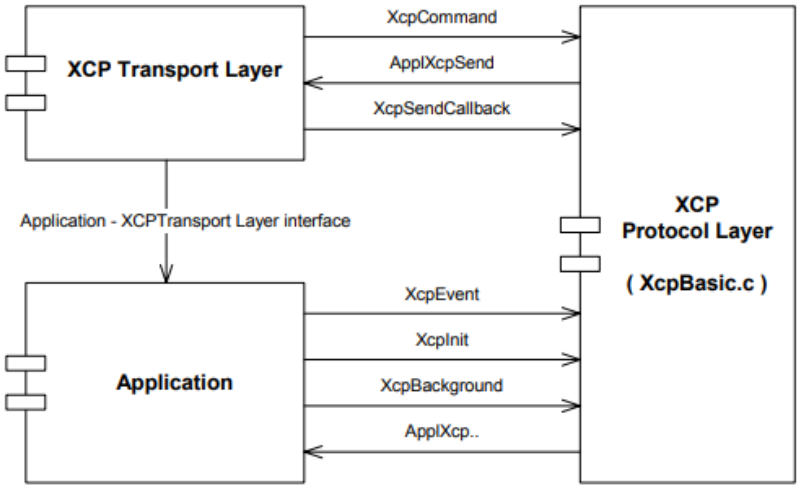

Figure 3.7: XCP communication model[27]

The application layer is responsible for managing and integrating both the XCP transport and protocol layers. It has to initialize the lwIP stack for enabling the transport layer, but also initialize the XCP protocol layer by calling the ***XCPInit*** function.

XCP Protocol Driver

The XCP protocol driver is integrated into the project with the following files:

- XCP_cfg.h
- XCP_def.h
- XCP_par.h/.c
- XCPBasic.h/.c

The configuration file ***XCP_cfg.h*** contains the preprocessor directives for configuring the driver and XCP settings. The Parameter files ***XCP_par.h/.c*** and Default settings file ***XCP_def.h*** are fine by default and do not have to be changed. The transport layer UDP/IP is implemented in the ***udpXCP.h/.c*** files. ***ipStackAurix.c/.h*** contains functions for the initialization of the lwIP stack and polling of received Ethernet packets. ***iXCPTargetPlatform.c*** is used by the protocol layer to call specific ECU, application, and transport layer functions. It contains specific callback functions, which implement the stub functions from the protocol driver [29]. ***XCPTask.h/.c*** files contain functions for the initialization of the complete XCP driver.

lwIP Stack

As mentioned, the basis of the Ethernet driver in this project is the lwIP stack, which is specially developed for embedded systems. It is open source and included in the "AURIX TC277x TFT Application Kit Firmware V1.1". It uses the consumer-producer paradigm for communication. Thereby, the producer is the Ethernet IRQ, which puts the received Ethernet packets into a queue. A function that is called cyclically in an OS task consumes this queue by polling it for received packets [13].

The lwIP stack is initialized in the main.c file at startup. This process includes setting the MAC- and IP-addresses. In the next step, the network stack is configured for a UDP/IP connection, and a Protocol Control Block (PCB) is being allocated in the memory. This PCB is then linked to a local IP address and a port. In the next step, a callback function for the PCB is registered, which will be executed if the PCB receives a UDP datagram.

For receiving Ethernet packets, the lwIP queue is begin polled in an OS Task. In the case of an incoming packet, the registered "receive-function" is called. In this function, the remote IP address and port are saved, and the UDP payload is mapped on a data-structure, that represents the XCP-packet structure [29]. The XCP payload is then forwarded to the XCP protocol layer by calling the function ***XCPCommand***. This function evaluates the XCP packet and interprets the XCP command [27].

For sending Ethernet packets, the XCP protocol layer calls the ***ApplXCPSend*** function. The ECU and transport layer specific implementation of this stub function is the ***udpXCPSend*** function. The input for this function is the XCP payload and the size of the payload. It then adds the Ethernet specific XCP header to the XCP payload and copies both the header and payload in an XCP-Ethernet-frame data structure [29].

For sending CTO commands, the XCP-Ethernet-frames have to be sent in a single UDP datagram each. This is needed because the slave must send a response to every master request individually. Therefore, the protocol layer calls the function ***ApplXCPSendFlush*** to send each XCP packet separately.

3.2.4 XCP-Master Implementation

In order to access other ECUs memory and manipulate its data, the RPS needs to have an XCP-Master Controller. This is the core functionality of this project. With the XCP-Master Controller, the RPS is able to connect to any XCP-Slave ECU and perform measurements and calibration of variables in its memory, as well as bypass whole functions for testing of new implementations.

For this project, not all functionalities of a full XCP-Master implementation are needed. Only those which are required are implemented in the driver. Those functionalities include connection, polling, calibration, and DAQ measurement configuration. This set is enough for having RPS functionalities enabled.

For communication between the RPS and a target ECU, CAN has been chosen. CAN is still the standard communication interface in vehicle ECUs,

and most of the ECUs have an XCP-Slave on CAN implementation in them. Following this, a CAN transport layer for the XCP-Master Controller has also been implemented.

CAN Transport Layer

The CAN transport layer is implemented in ***CanXCP.h/.c*** files. For interaction with the CAN module of the Aurix TC277 Evaluation Kit, Infineon provided the Basic Software within the firmware framework. The files ***MULTICAN_1.h/.c*** include everything needed for initializing the whole CAN module, including the baud rate, CAN nodes, CAN message objects, and Interrupts. It also includes a function for triggering the transmission of CAN messages over the bus.

For setting up the CAN node, it is necessary to define the Rx and Tx Pins for connecting to the CAN bus. The pins for CAN communication are defined in the Aurix TC277 user manual, and they have been accordingly mapped in CAN node settings of this file. Afterward comes the configuration of the CAN message objects. One message object is defined for the outgoing messages and one for the incoming messages. For successful communication, the baud rate and message IDs have to be configured to comply with the target ECU. The baud rate is configured within the CAN node configuration, while the message IDs are configured within the CAN message object configuration. The message IDs for transferring XCP messages over the CAN bus in this project are 610 for the messages from master to slave and 61A for the messages from slave to master. The interrupts which will be triggered in case of a successful message transfer or receipt are also defined within the CAN message object configuration. For this project, the interrupt for successful message transmission is not needed, but the interrupt for receiving CAN messages is of great importance. The role of the CAN RX interrupt is explained in later sections.

For handling of the data bytes to be sent/received over the CAN bus, there are two 8-byte message structures defined: one for the received messages (rxMsg) and one for the transmitted messages (txMsg). The structures are further divided into two 4-byte variables: data[0] and data[1]. Data[0] represents the four higher significant bytes of the data field, while data[1]

represents the four lower significant bytes. Figure 3.8 shows this concept for transmitting messages. Thereby, ***g_multican*** is the global handler variable for the CAN module.

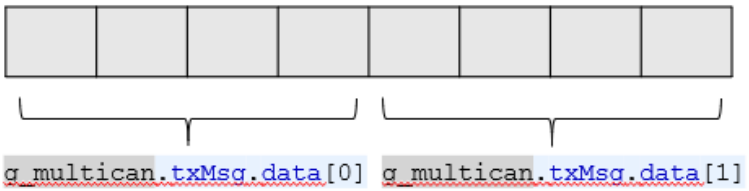

Figure 3.8: CAN data bytes handling

The state machine of the XCP-Master Controller is managed by an OS Task and the CAN RX interrupt handler. All functions of the driver are implemented in the ***XCP_Can_Master.h/.c*** files. It includes functions for connecting to an XCP-Slave, sending poll and calibration commands, as well as configuring a DAQ transfer from slave to master. The initialization of the XCP-Master Controller is performed within an OS task by invoking the function for sending the command for connection to the slave. This is always the first step. Afterward, based on what working mode is selected, further commands are sent. Data acquisition can either be in the DAQ mode, or by polling every individual variable.

The function ***CanIsrRxHandler*** in ***MULTUCAN_1.c*** contains the main code for controlling the state machine of the XCP-Master. Since XCP works based on the command-response mechanism, after each command the master sends, it has to wait for a response from the slave before sending a new one. In order to have minimal overhead and fastest possible communication, the CAN RX interrupt handler is used for triggering the processing of receiving responses from the slave and control of the program flow. Each time the RX interrupt is triggered, it also triggers the next step in the program. The program flow is controlled by a switch-case statement and proper flags.

Message Handling

CAN message sending is triggered by XCP-Master Controller functions. For that purpose, they embed the data to be transmitted into the two structures ***g_multican.txMsg.data[0]*** and ***g_multican.txMsg.data[1]*** and invoke the ***transmitCanMessage*** function. Thereby, it is necessary to embed the bytes in the correct order, so that the XCP-Slave receiving the message can interpret it.

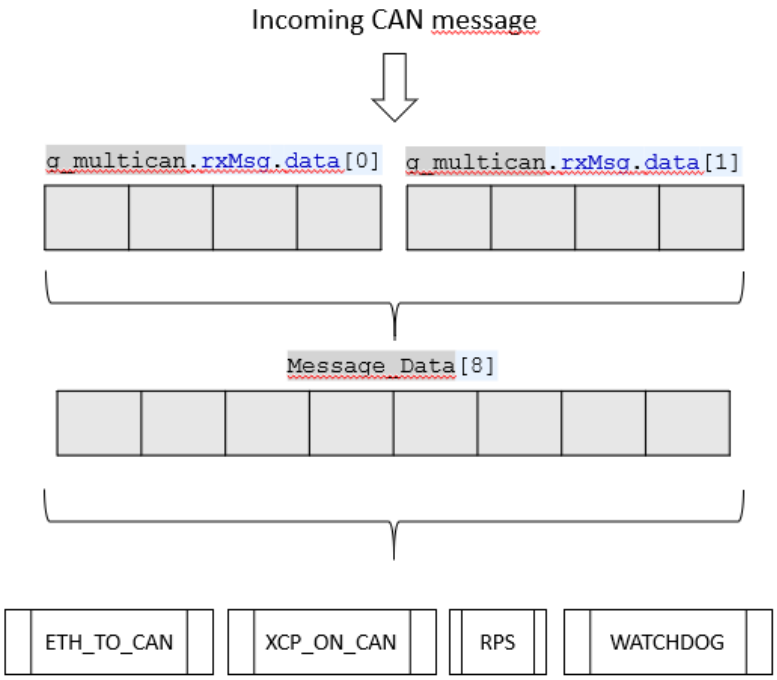

Figure 3.9: Handling of incoming CAN messages

In case of receiving CAN messages, the CAN RX interrupt handler is responsible for handling the messages. The activation of the interrupt triggers storing of the incoming CAN message payload into two structures ***g_multican.rxMsg.data[0]*** and ***g_multican.rxMsg.data[1]***. Afterward, all individual bytes of those two structures get mapped into an 8-byte structure,

thus enabling easy handling and interpretation of each individual byte. Based on the application, this structure is then forwarded accordingly to specific software functions for interpretation. Figure 3.9 shows this process.

Initialization of XCP-Master Controller

The XCP-Master Controller gets initialized in the OS Task by calling of the ***XCPM_Connect*** and ***XCPM_start*** functions. Prior to that, it is necessary that the CAN module is initialized and operational, which is performed during the startup of the board.

The ***XCPM_Connect*** function is used for establishing a connection with an XCP-Slave by sending the three default connection commands. After a successful connection, the OS Task will trigger the ***XCPM_start*** function. This function initializes the buffers for storing information about the signals to be measured and parameters to be calibrated.

In order to access a variable in an XCP-Slave for measurement/calibration, it is necessary to know the memory address location and the size in bytes of that variable. Therefore, there are two buffers defined which contain information about the memory address, length in bytes, and value of the variable: They are named ***Signal*** and ***Parameter***. The ***Signal*** buffer contains information about the signals to be measured in an XCP-Slave and the values of those signals. The ***Parameter*** buffer contains the same information but about the parameters to be calibrated and calibration-enabling switches in an XCP-Slave. Those ”switch” variables need to be set accordingly in order to enable the usage of the calibrated variables within the target ECU. If they are not enabled, the calibrated values in the target ECU will be ignored. After the initialization of the buffers, a specific function is executed, which will trigger the next step based on the application.

3.2.5 Rapid Prototyping System

The XCP-Master Controller provides the possibility to connect to an XCP-Slave and read/write to its memory. The combination of measurement and

calibration processes over XCP enables bypass of functions in XCP-Slaves and thus the usage of the platform as a rapid prototyping system. The inputs of the function to be bypassed are being ”measured” by the RPS and fed to a bypass function that is running on the RPS itself. The outputs which the bypass function produces in the RPS are then used for calibration of the output values of the function to be bypassed. The target ECU can now use the calibrated values and not those calculated by its internal function. This way, new algorithms can easily be integrated into a real system and evaluated. Figure 3.10 shows the concept of RPS utilization.

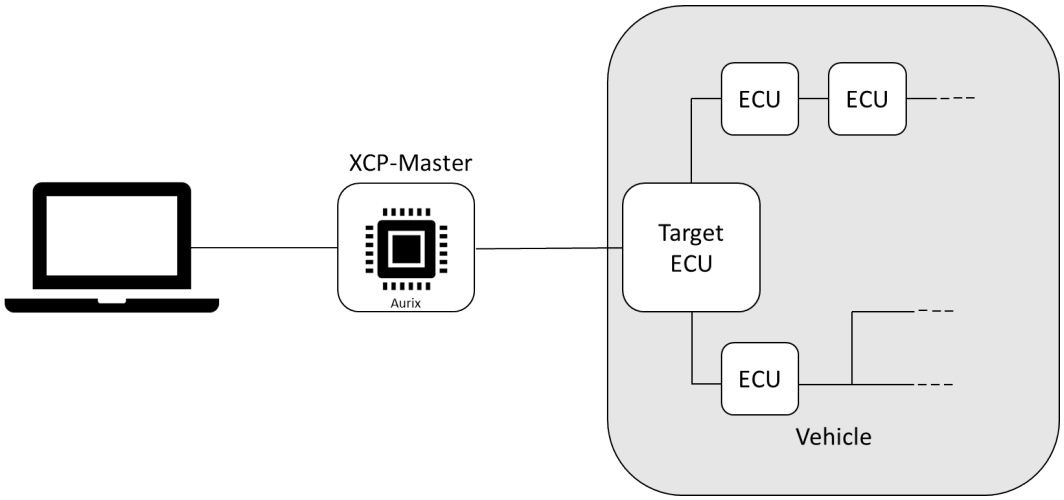

Figure 3.10: Rapid prototyping system concept

The header file ***XCP_CAN_master.h*** configures the working mode of the platform by preprocessor directives. In order to have RPS functionalities, ***XCP_MASTER_RPS*** has to be defined there. This will activate all necessary components in other software modules during compiling, which are needed for RPS functionality. After the initialization of the XCP-Master Controller (connecting and buffers initialization), the next step is to configure the measurement process. There are two available measurement options: polling and DAQ.

If polling should be used as the measurement process, no additional configuration is needed. Measurement commands are sent right after the initialization of the XCP-Master Controller. If DAQ should be used, the next step is

to configure the XCP-Slave to send the required data via DAQ. The process of DAQ configuration includes sending a significant amount of commands, which is described in the DAQ Configuration subsection below. When the measurement process is configured and all measurements are received by the RPS, a flag indicating successful measurement reception gets set.

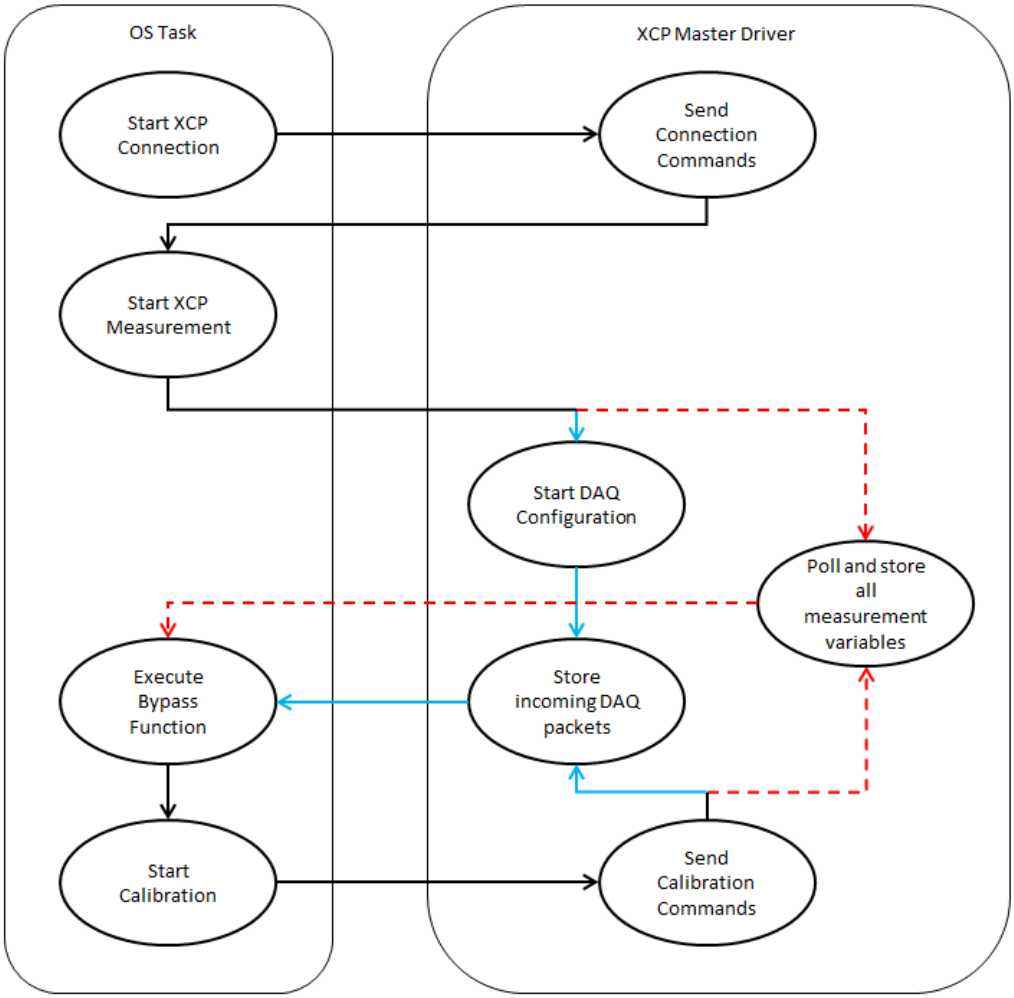

Figure 3.11: State Machine of the XCP Master Controller

An OS Task is monitoring this flag, and in case it is activated, it forwards the measurement signals to the bypass function and executes it. The algorithm

of the OS Task is shown in figure .1 in the appendix. After the bypass function has produced the output parameters for calibration, those values need to be transferred to the target ECU. Thus, the calibration process is being started.

For calibrating a value over XCP on CAN, two commands have to be sent. One commands sets the memory address of the variable to be calibrated, and the other command sends the size in bytes and value of the variable. After the calibration is performed, the RPS goes back to waiting for the next measurements and repeats the cycle with each new measurement set.

The state machine is shown in figure 3.11. The red lines indicate the process flow when DAQ is chosen as the measurement method, while the blue dashed lines indicate the process flow with Polling as the measurement method.

For controlling the program flow, a switch-case statement in the CAN RX interrupt is being used. The variable used in the switch-case statement is called ***ProcessPhase***. This variable can be in 5 different states:

- XCP_CONNECT
- DAQ_CONFIGURATION
- MEASUREMENT
- DAQ_MEASUREMENT
- CALIBRATION

Based on what operating mode is selected, the ***ProcessPhase*** variable changes its value to ensure the appropriate program flow.

Calibration Switch-Variables

Automotive ECUs have a software mechanism that enables the usage of bypass variables instead of internal calculated ones. Based on the status of the corresponding switch-variable, the internal program flow uses either the bypass or the calculated variable. Therefore, in order to calibrate any values in the target ECU, the corresponding switch-variables have to be set to enable the bypass variables. This is performed by sending calibration commands to the target ECU for setting those switch-variables to be TRUE.

The commands for enabling the switch-variables are going to be sent during the first calibration phase.

Calibration

After the bypass function has generated the output variables which have to be transmitted into the target ECU, the calibration process is started. The variables are calibrated one by one in a sequence of commands. In order to calibrate a variable, an XCP-Slave needs to know the address of the variable, its size, and its value. Since one CAN message can transfer only eight useful bytes, it is not possible to transfer all the information with one message. Therefore, for calibration of a parameter, two messages are sent. Each message encapsulates a different XCP command. The first one is the XCP command SET_MTA and the second one is DOWNLOAD. SET_MTA informs the XCP-Slave about the address of the variable to be calibrated, while DOWNLOAD informs about the size and value of the variable. After successful transmission of those two commands, the variable will be calibrated in the memory of the XCP-Slave.

Thereby, the CAN RX interrupt is acting as a loop. This is because, after every sent command from master to slave, a response has to be received from slave to master before proceeding to send the next command. Using the RX interrupt here as a loop ensures the fastest possible communication. For controlling the order of the messages, two counters are being used: ***CalibCounter*** and ***MessageBuffer***. ***CalibCounter*** counts all sent messages, and its value is going to be twice as high as the number of calibration variables. This is because, as mentioned, every calibration variable needs two messages to be successfully calibrated. When all calibration messages are sent, ***ProcessPhase*** switches to another phase, and the counters get reset. ***MessageBuffer*** counter is used for ensuring the consistency of the two messages for calibrating one variable. Both messages need to embed data from the same Parameter buffer location. This counter gets increased after the successful calibration of one variable.

Before sending a calibration command, it is checked if the value to be calibrated is changed compared to the previous step. If so, then it will be sent for calibration, but if it has not changed, then the ***MessageBuffer*** counter

increases, and the next variable in the Parameter buffer is checked. This way, unnecessary bus load is avoided. Furthermore, the switch-variables are also only set during the first calibration iteration and afterward, only if they are changed during run-time.

The algorithm of this implementation is shown in figure .6 in the appendix.

Connection

In order to communicate with an XCP-Slave using the XCP-Protocol, first, a connection has to be established. This is performed by sending three default commands from master to slave. If the connection is successful, the slave will send a positive response for every command, but also additional information about the capabilities of the XCP-Slave.

For controlling the order of the connection commands, the counter Connect-Counter is used. After each connection command, the XCP-Slave sends a response and activates the RX interrupt again. If the response is positive, the program flow gets into the loop again, increments the counter, and sends the next connection command. When all connection commands are sent, thus the counter reached its maximum value, the connection status flag ***XCPConnectStatus*** is set to CONNECTED, and the counter is reset. This indicates a successful connection, and the program flow can proceed to the next step based on the application.

The algorithm of this implementation is shown in figure .2 in the appendix.

DAQ Configuration

For configuring an XCP-Slave to send DAQ packets, a sequence of specific XCP commands has to be sent from master to slave. This way, dynamic DAQ lists are being generated and configured in the XCP-Slave. The command sequence is shown on figure 3.12.

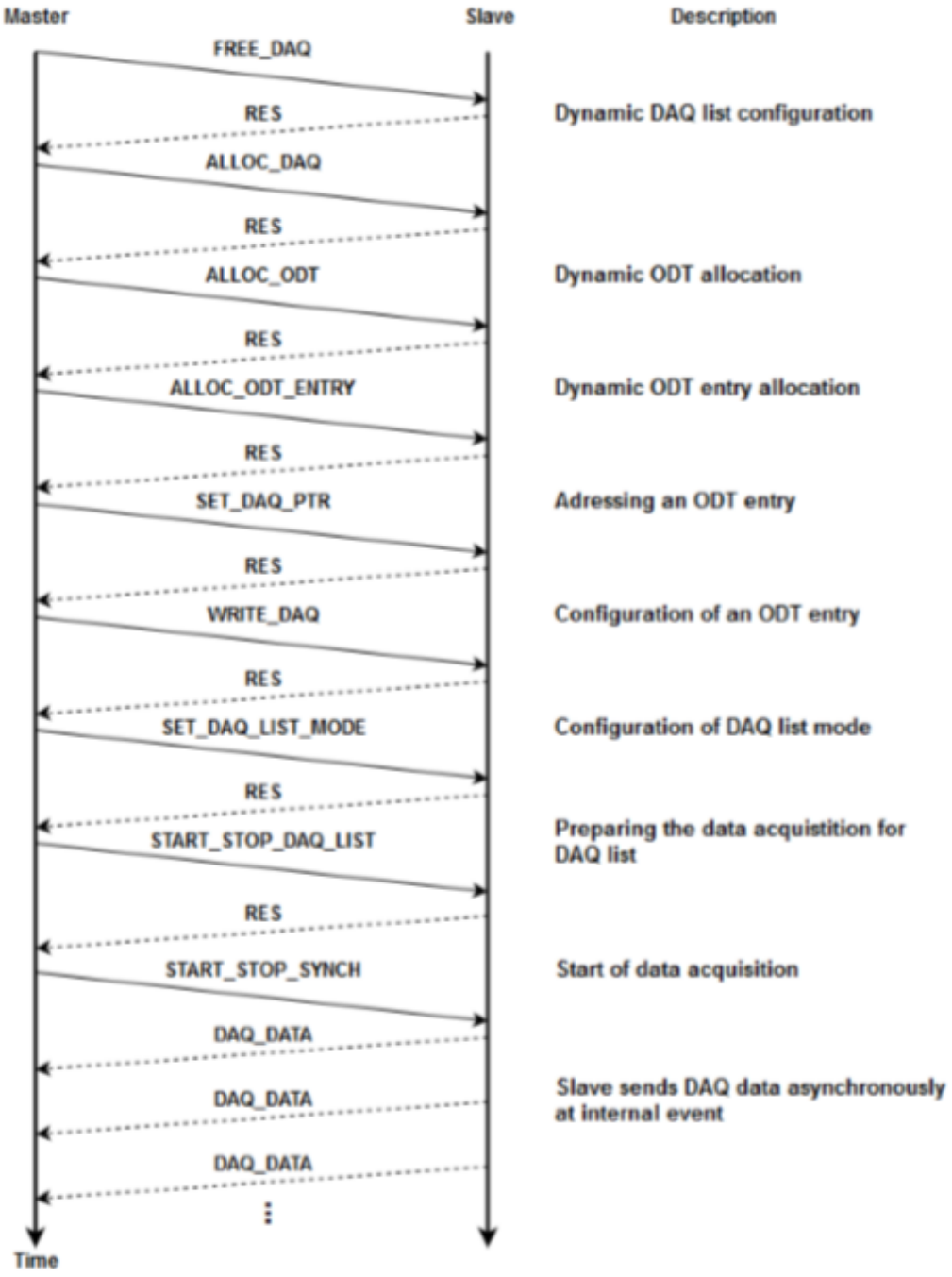

Figure 3.12: DAQ configuration sequence

The proper order of the commands is ensured with another switch-case statement and proper flag utilization.

The Counter ***DAQCounter*** is used as the variable for the switch-case statement and incremented accordingly to proceed to the next steps. In some cases, a function needs to be executed more than once. Those functions decrease the counter by one and keep it at the same value until the function is executed the needed amount of time.

The DAQ configuration sequence is started after the calibration configuration sequence is finished by calling the XCPM_Start_DAQ() function. This function sets the ***ProcessPhase*** to DAQ_CONFIGURATION and sends the first command in the chain (FREE_DAQ). During configuration of DAQ lists in an XCP-Slave, this command must always be sent first in order to clear all existing DAQ lists in the slave and make it ready for the configuration of new ones.

The next command to be sent is ALLOC_DAQ, which allocates a defined number of new DAQ lists in the slave. Following this is the ALLOC_ODT command. This command allocates Object Description Tables in the DAQ lists defined by the previous command. For each DAQ list, one command has to be sent, which defines the number of ODTs in that specific DAQ list.

Afterward, ODT entries are defined for every ODT in the previous step with the command ALLOC_ODT_ENTRY. For every ODT, one command has to be sent, which defines the number of ODT entries for that specific ODT of a specific DAQ list.

The command SET_DAT_PTR initializes the DAQ list pointer for subsequent operation with WRITE_DAQ or REA_DAQ. This command basically selects the DAQ list to be configured in the next step.

Following this, comes the WRITE_DAQ command. This command is used for configuring the size and addresses of all variables to be read from the slave. For Each Variable (ODT entry), one command is sent from master to slave. After configuring one DAQ list and proceeding to configure the next one, SET_DAT_PTR command is sent again to set the pointer to the next DAQ list. Then again, by the usage of WRITE_DAQ command, the size and

addresses of variables of the next DAQ list are configured. This process is repeated until all DAQ lists are configured.

Afterward, the SET_DAT_LIST_MODE command is sent from master to slave. This command defines the DAQ list priorities and to which internal event number of the XCP-Slave a specific DAQ list will be linked. Those events are usually defined to trigger every 5 ms, 10 ms, etc. If a DAQ list is linked to a 10 ms event, the slave will gather all data defined within that DAQ list and sent it to the master every 10 ms.

The START_STOP_DAQ_LIST command is used to start, stop, or prepare a synchronized start of the specified DAQ list number. This command is sent for each DAQ list individually. If a synchronized start is chosen, then the next command in line will start the DAQ transmission. The START_STOP_SYNCH command is used to start and stop synchronized DAQ list transmission. All DAQ lists which have been configured in the previous step for a synchronized start will start when this command is received by the slave. The transmission can also be stopped using this command.

In this project, one DAQ list is defined, which contains all variables to be measured. For every signal to be measured, one ODT with one ODT entry is defined. This ensures that every signal will be sent in one CAN message. This approach makes it much easier to interpret the incoming messages and track which message contains which signal value.

The algorithm of this implementation is shown in figure .3 in the appendix.

DAQ Measurement

After DAQ configuration is finished, DAQ data transfer is started by sending the START_STOP_SYNCH command from master to slave. From this point on, the slave will send DAQ data according to the configured DAQ lists. Every signal to be measured is sent by one CAN message. The first byte of the DAQ payload is the relative number of the message, while the rest of the bytes represent the value of the signal. The first message always starts with 00, and all next messages will have this value incremented by one compared to the previous one. This makes it convenient to handle the data

of the messages since the first byte identifies every message, and the order of Signals to be sent via DAQ is known.

Each message of the DAQ packet triggers the CAN RX interrupt, which stores the current payload it has received by executing the XCPM_StoreResult() function. This process is repeated for every DAQ message. The first byte of the payload is also used as a counter variable to handle the storing of message data. When the complete DAQ data is received by the RPS, a flag named ***SignalsReceived*** is set to TRUE. The 1 ms OS Task is checking this flag with every iteration, and if it is true, it disables interrupts and executes the Bypass function with the received DAQ data as the input. When the Bypass function is executed, and the output values (calibration data) are ready, the calibration phase is started by executing XCPM_Calibrate() function.

The algorithm of this implementation is shown in figure .5 in the appendix.

Measurement by polling

Unlike DAQ, measurement by polling does not require any additional configuration. With this method, every signal is polled for its value by an XCP command, called SHORT_UPLOAD. This command sends information about the address to be read, and the length of the variable. When an XCP-Slave receives this command, it gathers the data defined by the message and sends the value back with a message. This approach, therefore, requires constant message exchange between master and slave and loads the bus heavily compared to DAQ. However, it does not have any requirements on the XCP-Slave side, such as events with DAQ.

The principle of storing message data and calibration is the same as with DAQ. The only difference is how measurement data from the slave is obtained (the ProcessPhase variable switches to MEASUREMENT mode instead of DAQ_MEASUREMENT mode).

3.2.6 Intelligent Watchdog

For intelligent watchdog functionality, the first steps are the same as with the RPS: connect via XCP and configure the measurement. For a watchdog, it is very important that the measurement data is consistent. Therefore, only DAQ measurement method is suitable for watchdog functionality. This ensures data consistency. If polling would be used, we have no guarantee that the measurement data is from the same computation cycle of the ECU.

The configuration process is the same as with RPS. After the data is received, it is being evaluated according to the watchdog criterion. If the data seems to be faulty, an algorithm can be implemented in the watchdog to perform actions that will ensure that the process is still operational. It could either activate another redundant system or take over the execution of compromised functions in the target ECU.

3.2.7 Ethernet to CAN Interface

The UDP/IP and CAN transport layer provide the basis for the implementation of an Ethernet-CAN interface. Messages which are received in one layer can easily be processed in such a way, that the payload of the incoming message is extracted and embedded into the data frame of the outgoing message in the other transport layer. This way, messages are ”converted” from one to the other transport layer within the platform. Thus, an XCP session between a device with XCP over Ethernet can be established with a device with XCP over CAN.

This mode is activated by the defining the preprocessor directive ETH_TO_CAN_INTERFACE. This configures the UDP/IP message handling in udpXCP.c to execute the function Eth2Can() when a UDP packet has been received. This function receives the payload as an argument and embeds the payload bytes into the CAN specific frame. Afterward, the transmitCanMessage() function is called to transfer the message to the CAN bus.

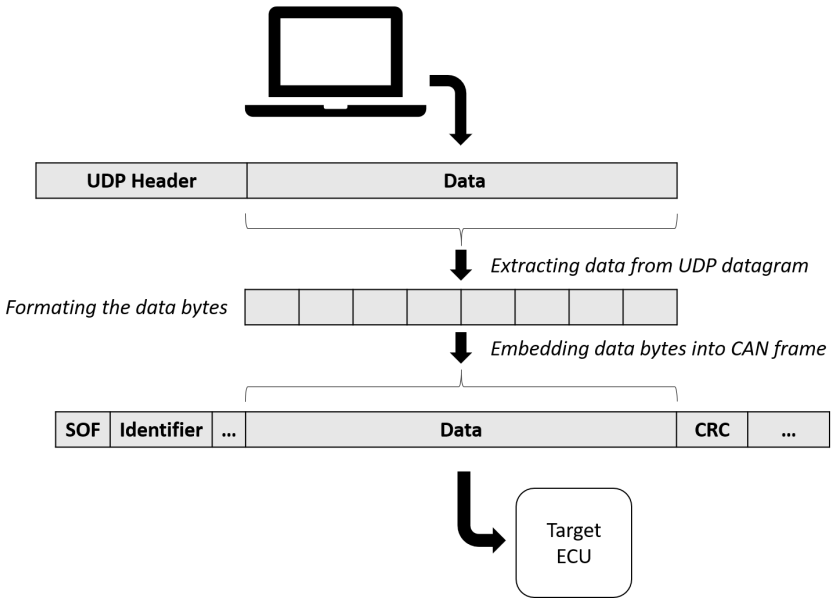

Figure 3.13: Ethernet-to-CAN message sequence

On the other way, when a CAN message is received, the CAN RX interrupt gets activated. In the Interrupt handler, the payload of the CAN message is being extracted and passed to the function udpXCPSend(). This function prepares the UDP/IP message with the defined payload to be sent via Ethernet. Figure 3.13 shows this process for Ethernet-to-CAN conversion. The principle of CAN-to-Ethernet is the same.

3.3 Application Configuration

For configuring the RPS to work with a specific ECU, a configuration tool in python has been developed. This application provides a graphical user interface (GUI) for setting up all required settings and generating .c and .h configuration files. It is shown in figure 3.14.

The basis for the configuration is the A2L file of the ECU. This file includes information about all functions and variables, as well as communication

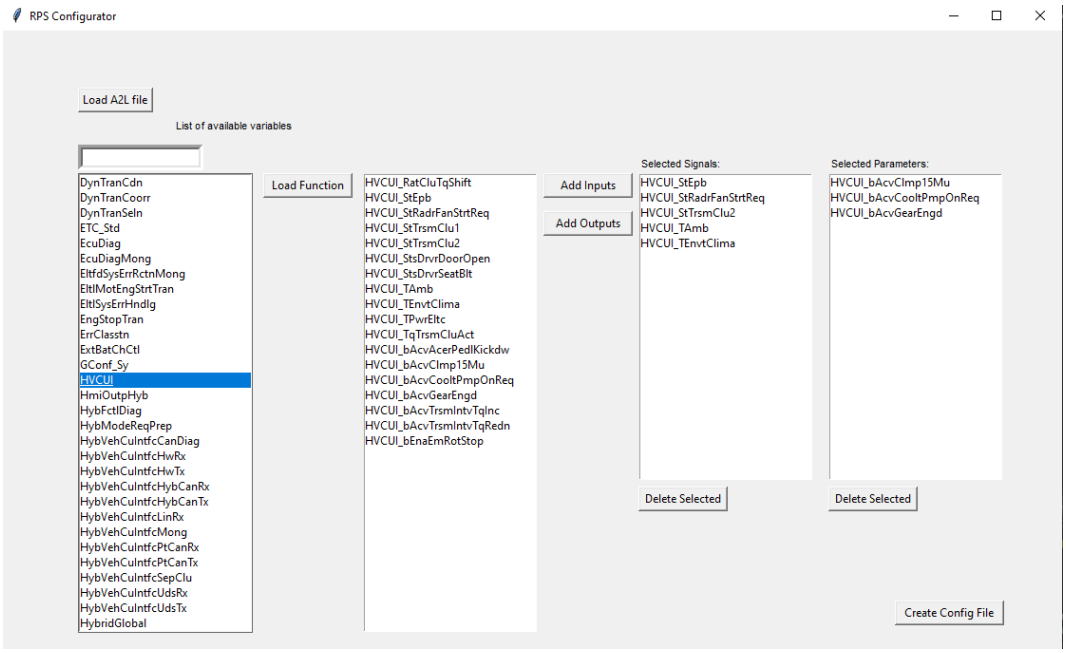

Figure 3.14: DAQ Configuration tool GUI

interfaces and XCP settings of the ECU. Hence, the first step is loading the ECU specific A2L file into the configuration application. After a successful A2L load, the first list box in the GUI will show all available functions which are found in the A2L file, hence the ECU. The user can now select a function of interest and load all variables connected to that function into the second list box. This is performed by selecting the desired function and pressing the ”Load Variables” button. In the second list box, the user can select multiple variables and add them to one of the next two list boxes representing the input Signals and output Calibration parameters. This is performed by either pressing the ”Add to Inputs” or ”Add to Outputs” button.

Those two list boxes represent all variables that will be included in the final configuration .c and .h files and linked to the buffers explained in section 3.2.4. Thereby, the Output variables include not only the calibration variables but also the switch-variables of the ECU, which enable usage of calibration variables and are explained in chapter 3.2.4. If some variables

have been added by mistake, they can be selected and removed from the list by clicking the ”Remove Variable” button underneath the list box. When all variables of interest are imported into these two list boxes, the button ”Create Configuration File” will trigger the process of creating the .c and .h files for configuration.

These files are generated in two steps. The first step after the ”Create Configuration File” buttons is pressed, is the generation of an excel file. This excel file contains two sheets, one for input Signals, one for output Calibration parameters. Every sheet contains information about the variables which are selected by the user during the configuration process in the GUI. That information is the name of the signals, data types, memory addresses, and sizes in bytes. After this step, this excel file is used to generate the final .c and .h files. This approach provides the possibility to skip the GUI configuration process if the user already has all the information required in the excel file. The user can simply add all variables manually into the excel sheets and generate configuration files based on that.

The files which are generated are:

- RPS_configuration.c
- RPS_configuration.h
- RPS_interface.h

RPS_configuration.c contains declarations of the input and output signals as global variables, as well as the declaration of functions for initializing the buffers, and for linking them with the input and output variables. RPS_configuration.h contains the declarations of function prototypes from RPS_configuration.c. RPS_interface.h is the file which the user should use to establish the interface between the RPS and his bypass function. It contains an external declaration of the input and output signals so that the user can link them with his bypass function variables. Furthermore, it includes the prototype of the function RPS_Bypass_10ms(). RPS_Bypass_10ms() is used for linking the measured input signals from the RPS with the input signals of the bypass function, execution of the bypass function, and afterward linking the output variables of the bypass function with the output variables of the RPS, which will be sent for calibration. This function should be defined by the user, and an example of how it should look is provided in comments at

the end of the RPS_interface.h file. All the configuration files are stored in the same folder as the python script.

4 Test Environment and Evaluation

This chapter evaluates the project implementation and discusses the obtained measurement result. It describes the test setup and methods used for proving the implemented concepts, as well as the obtained measurement results. Furthermore, it also provides descriptions and observations of the measurement result figures and CAN message logs. As the test subject, a real automotive ECU is used from an AVL project. It offers an XCP on CAN support.

4.1 Test Environment

4.1.1 Rapid Prototyping System

For evaluating the RPS implementation, the ECUs A2L file is the entry point. It is used together with the configuration tool to configure the RPS accordingly, as described in chapter 3.3. For RPS demonstration purposes, a function for regulating the duty cycle of a pump is chosen to be the bypass function. As the source code of the ECU is available, the same function is integrated into the RPS.

As both the RPS and the test ECU execute the same function, measurements show a comparison between the RPS-calculated values and the internal ECU-calculated values. The function's main input signal is the oil temperature. For simulating the rise in oil temperature, a for-loop is implemented in the RPS, which generates temperature values from 0°C to 100°C in a cyclically defined step.

The oil temperature is then written into the ECU via XCP calibration so that it can be used as the input of the duty cycle function within the ECU (Number 1 in figure 4.1). Furthermore, as the RPS gathers signals from the ECU, it will also obtain and use the oil temperature value, which was calibrated into the ECU by the RPS itself (Number 2 in figure 4.1). This ensures that the RPS uses only signals obtained from the ECU as input to its functions.

Based on those input signals, the RPS executes the bypass function and generates the function outputs (Number 3 in figure 4.1). Those outputs are then written to their corresponding memory locations into the ECU (Number 5 on figure 4.1). During the first calibration cycle, the corresponding calibration switch-variables in the ECU are set to be enabled, as described in chapter 3.2.4 (Number 4 in figure 4.1). This ensures that the internal algorithm of the ECU uses the calibrated values of a variable instead of the internally calculated one.

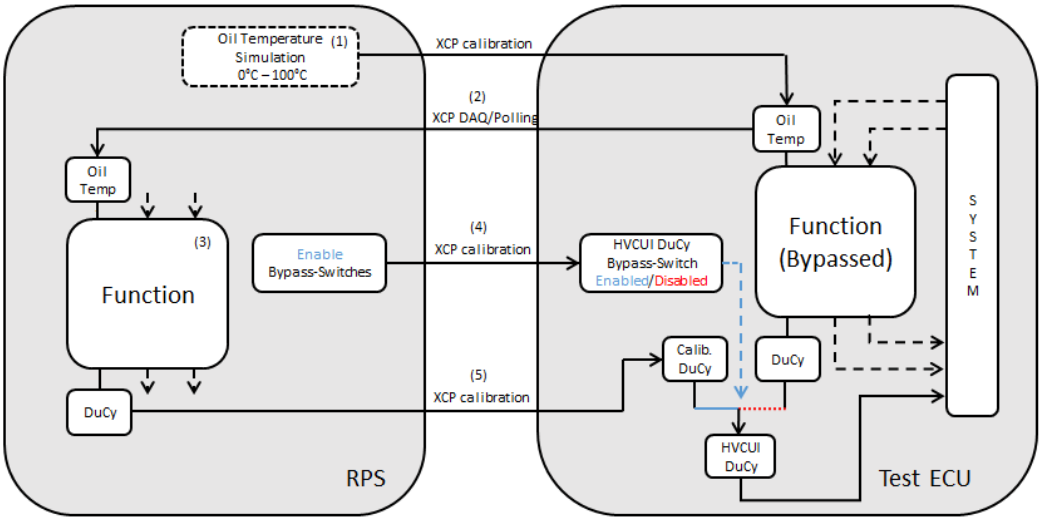

Figure 4.1: Bypass Concept for Evaluation

A debugger is connected to the ECU for signal logging needed to compare the internally calculated values of the ECU and the values calculated and

calibrated by the RPS. Besides the calculated and calibrated duty cycle values, it also logs the oil temperature stimulated in the ECU. Measurements are performed with both the DAQ and the polling method.

For evaluating the RPS with DAQ implementation, two measurements are performed. During the first measurement, the RPS executes the same function as the test ECU and calibrates the output variables it calculates into the ECU. The ECU also calculates the same output variables, but those values are bypassed by the calibrated ones since the calibration switch-variables are enabled.

During rapid prototyping, sometimes it can happen that only a signal value within the ECU needs to be corrected by a factor. In order to demonstrate this use-case too, the bypassed duty cycle value is set to be 1.1 times higher than the internal value of the ECU.

The setup for evaluating the RPS with Polling implementation is the same as with DAQ, but only that Polling is used as the data acquisition method. The bypassed variable is scaled by a factor of 1.1, and the bypass is enabled at a point of time during the measurement.

4.1.2 Intelligent Watchdog

The intelligent watchdog functionality is evaluated with the same hardware setup as for the RPS evaluation. The Watchdog-implementation is configured to execute the same function as the ECU. In order to check if any errors or faults occurred during the ECU operation, it reads the variables to be monitored from the ECU. For that purpose, the data to be monitored by the watchdog is configured for a DAQ transfer from the ECU. This setup-initialization is performed during the startup of the platform.

For the demonstration, the variable to be monitored by the watchdog is chosen to be the cooling pump duty cycle. As the duty cycle depends on the input oil temperature, it is again simulated by the platform and written into the ECU via XCP calibration. After receiving the measurement data from the ECU, the watchdog executes the same function as the ECU and calculates the duty cycle on its own. Afterward, it executes a comparing

function in order to determine if the values from the ECU differ from the values calculated by the Watchdog itself. If the difference of those two values exceeds a predefined limit, a counter (see Watchdog Counter on Figure 4.7) is increased and keeps increasing for every cycle that the allowed difference is exceeded. If this counter reaches a critical value, the watchdog triggers a fault reaction. Furthermore, if the value difference goes back to normal, the counter gradually decreases until it reaches the critical value again, after which the watchdog fault reaction is reset.

In order to simulate a fault in the ECU, the signal value which is obtained by the ECU is held at a constant value at a point of time. At a later point, this behavior is again reset, so that the watchdog recovery could be observed.

4.1.3 Ethernet-CAN Interface

For evaluation of the Ethernet to CAN interface, bus load measurements on both CAN and Ethernet side are performed. To establish high message traffic on the CAN bus, a transfer of a high amount of data is configured. This is performed with CANape as the XCP master and the project ECU as the XCP slave. For measuring the CAN busload, "bus master"[1] software is used, and for measuring the Ethernet busload, "WireShark"[2] software is used. Two measurement sets are performed, each including 4-byte and 1-byte signals.

One set is performed with DAQ transfer mode while the other set is performed with Polling. The amount of signals to be transferred is gradually increased in each measurement step so that a correlation between the number of signals and the thereby caused bus load could be observed. DAQ and Polling measurements are configured to occur cyclically at the 10 ms intervals.

[1] www.etas.com/en/applications/applications_busmaster.php
[2] www.wireshark.org

4.2 Evaluation Results

4.2.1 Measurement 1: RPS with DAQ

Figure 4.2 depicts a comparison between the RPS values and ECU values without any scaling.

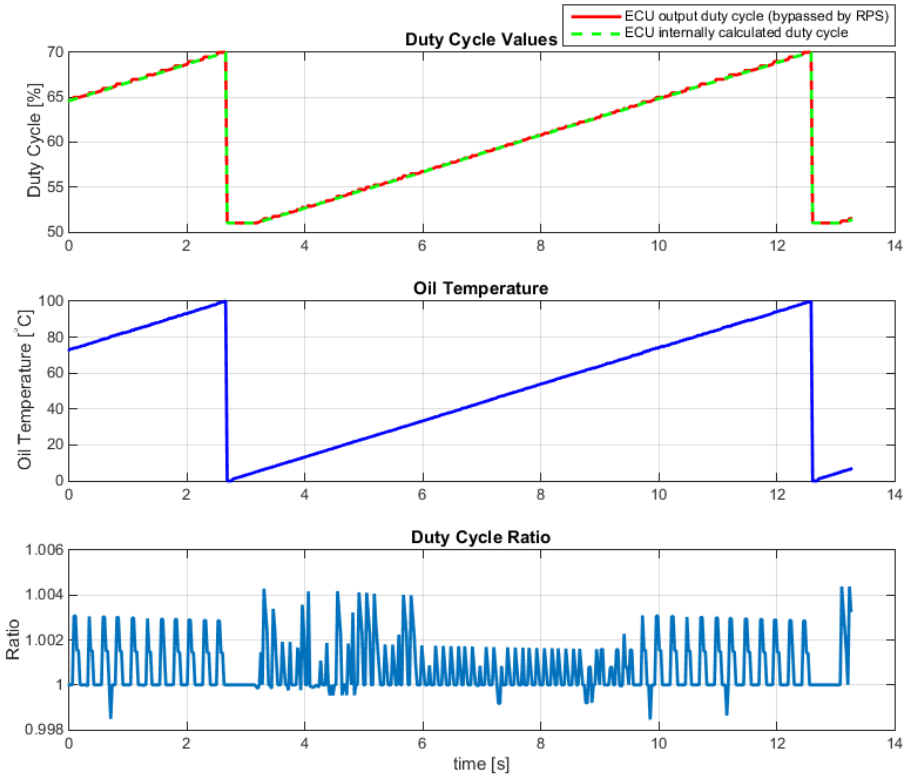

Figure 4.2: Direct bypass of a variable by using DAQ

The measurement results during bypass of the function and scaling of the duty cycle are shown in figure 4.3. Furthermore, the calibration switch for enabling bypass of the duty cycle is triggered at a point in time during the measurement process in order to observe the switch to the calibrated value.

The CAN messages exchanged during the whole process are logged with a PCAN adapter and software. The messages of one measurement and calibration cycle between the RPS and the ECU are shown in table .1 in the appendix.

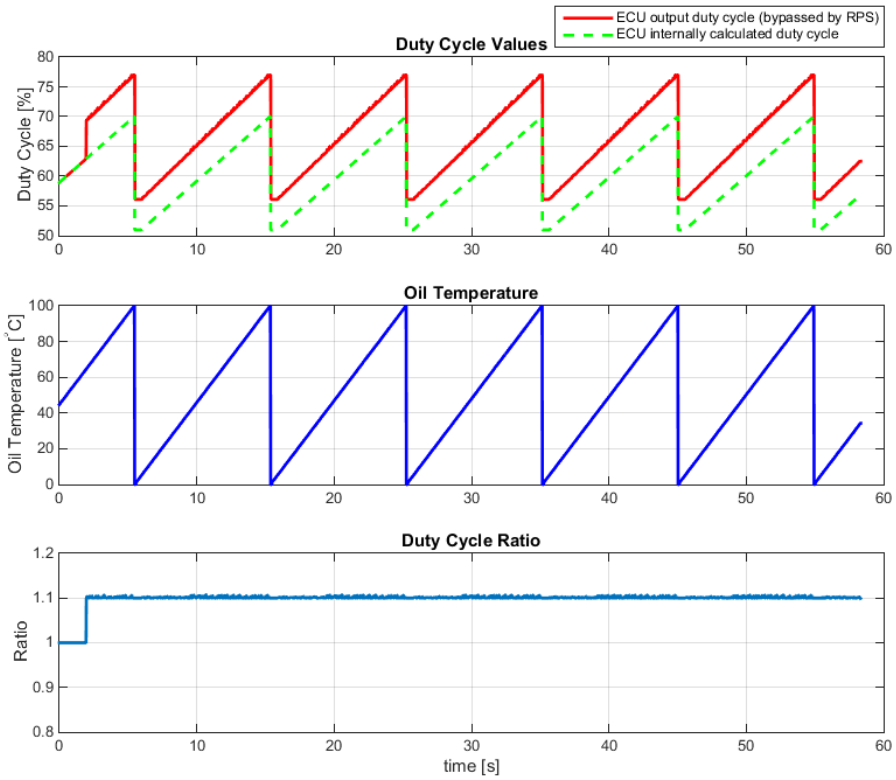

Figure 4.3: Bypass of a variable with a scaling factor of 1.1 by using DAQ

In order to check the consistency of the data sent via CAN bus and measured data on the ECU side, the CAN messages that contain the oil temperature values are extracted from the PCAN message log and shown in figure 4.4.

As a real-world automotive ECU is used from an AVL project, it provides a sound test object for measurements. Since the RPS implementation is limited to tasks with a minimum cycle time of 10 ms, the test function to be bypassed is chosen to be a temperature handling one. After researching this

particular ECUs documentation, a pump duty cycle calculation function that uses the oil temperature as the main input is selected as the test bypass function.

The data gathering method of the first RPS measurement is set to be DAQ. As the source code of the ECU is available, it is easy to integrate the ECU-function into the RPS. Figure 4.2 shows the measurement of the stimulated oil temperature (blue), the duty cycle calculated by the ECU (green), and the final output duty cycle value (red). The final output of the function can either obtain the ECU-calculated value or a calibrated value, depending on the state of the corresponding calibration switch variable inside the ECU.

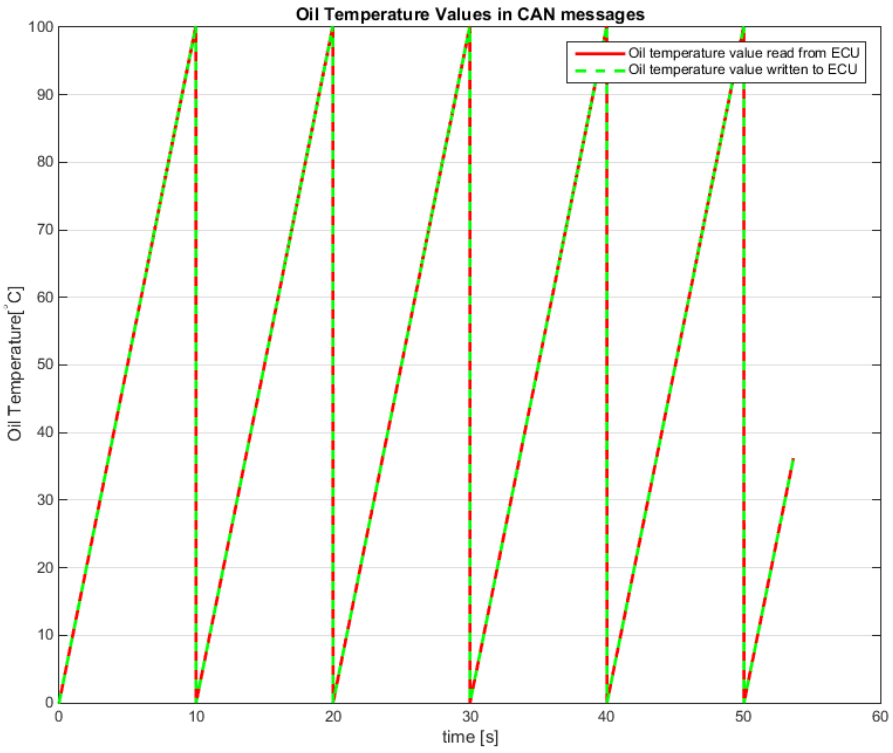

Figure 4.4: Comparison of the Oil temperatures sent and received via CAN during DAQ

Since the calibration switch variable for this signal is set to be enabled at the

very beginning of the measurement, the final output value of the function is also the duty cycle value calibrated by the RPS. It can be observed in figure 4.2, that two duty cycle values of the ECU and the calibrated values from the RPS are overlapping. The oil temperature is set to increase linearly from 0 to 100°C, as observable in the same figure, after which it restarts. The duty cycle also increases linearly with the oil temperature, but with a different scaling factor. The last plot of figure 4.2 shows the ratio between the final output value of the duty cycle function and the internally calculated duty cycle value. It can be observed that the ratio is in the range of 0.998 to 1.004. This deviation is caused by floating-point arithmetic and rounding errors.

The second test utilizes the same test setup but with a slight modification of the RPS bypass function. The duty cycle value to be calibrated by the RPS is set to be scaled by a factor of 1.1 compared to the default value, and the calibration is enabled at a point of time during the measurement. Figure 4.3 shows the measurement results obtained during this test. The oil temperature rises again from 0 to 100°C in a linear manner. At the very beginning of the measurement, the ECU-calculated duty cycle values and the final function output have the same value, as calibration is still not enabled (calibration switch variables not set). A short amount of time upon start, calibration is enabled, and now the final output takes the value of the RPS-calibrated one. It can be observed that the value is higher by a factor of 1.1 than the ECU-calculated one, as is expected. When the oil temperature value changes from 100 to 0, the internally calculated duty cycle obtains a new value before the new RPS value gets calibrated, and hence the ratio gets much higher during that process.

The CAN messages during one calibration and measurement cycle from table .1 show the speed of each message on the CAN bus. Since DAQ mode is configured as the data gathering method, all signals are sent cyclically by the ECU itself, as described in the "DAQ Measurement" section of the chapter 3.2.4. It can be observed that 12 DAQ CAN messages containing 12 signal values are obtained within a time window of 0.9 ms. After obtaining the last DAQ value, the bypass function is executed, and calibration is started. More information about the exact sequence and XCP on CAN message meaning used in this example can be found in chapter 3.2.3 and 3.2.4. Since only the changed values are being sent for calibration, in the cycle shown here, there are two signals present: the stimulated oil temperature and the

RPS-calculated duty cycle. It can be observed that the calibration of the two variables takes 0.8 ms.

Figure 4.4 shows the oil temperature values which are sent and received via CAN bus. The red signal is the oil temperature obtained via DAQ from the ECU, and the green signal is the stimulated value from the RPS. The figure shows that the signals are overlapping with each other.

4.2.2 Measurement 2: RPS with Polling

Figure 4.5 shows the obtained signal values during the measurement.

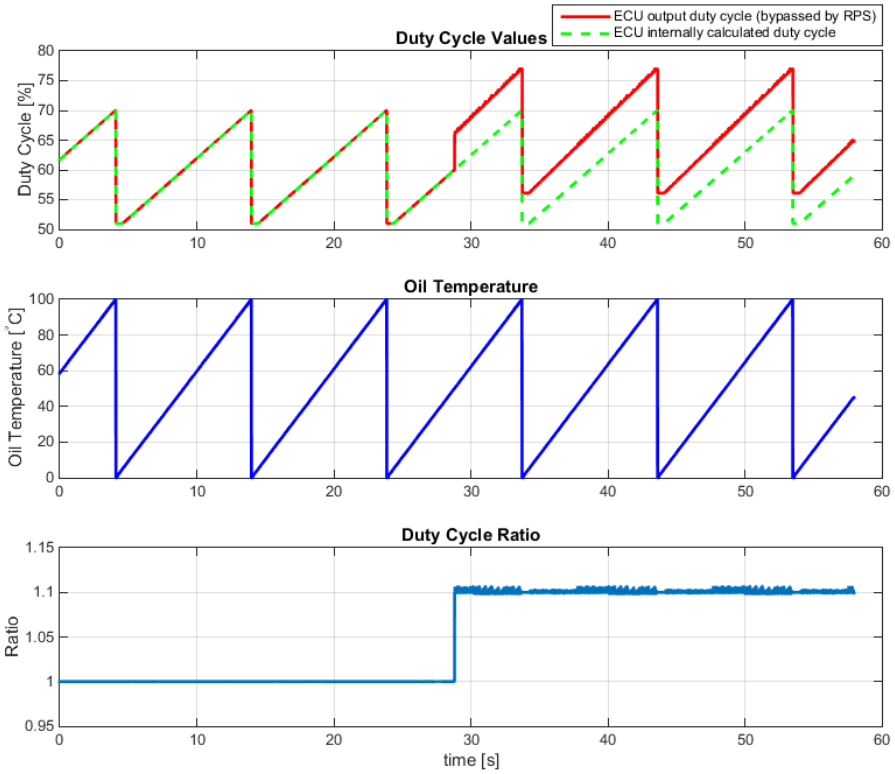

Figure 4.5: Direct bypass of a variable by using Polling

For checking the consistency also during polling, the Oil temperature values sent and received via CAN are extracted from the PCAN message log and shown in figure 4.6.

The CAN messages exchanged during one calibration and measurement cycle by using polling are shown in table .2 in the appendix. The test setup remains the same, and only the data acquisition mode is changed. Figure 4.5 shows the measurement results of the oil temperature (blue), ECU-calculated duty cycle (green), and the final function output (red). During this measurement, a scaling factor of the calibrated duty cycle value is also used, and calibration is enabled at a point of time.

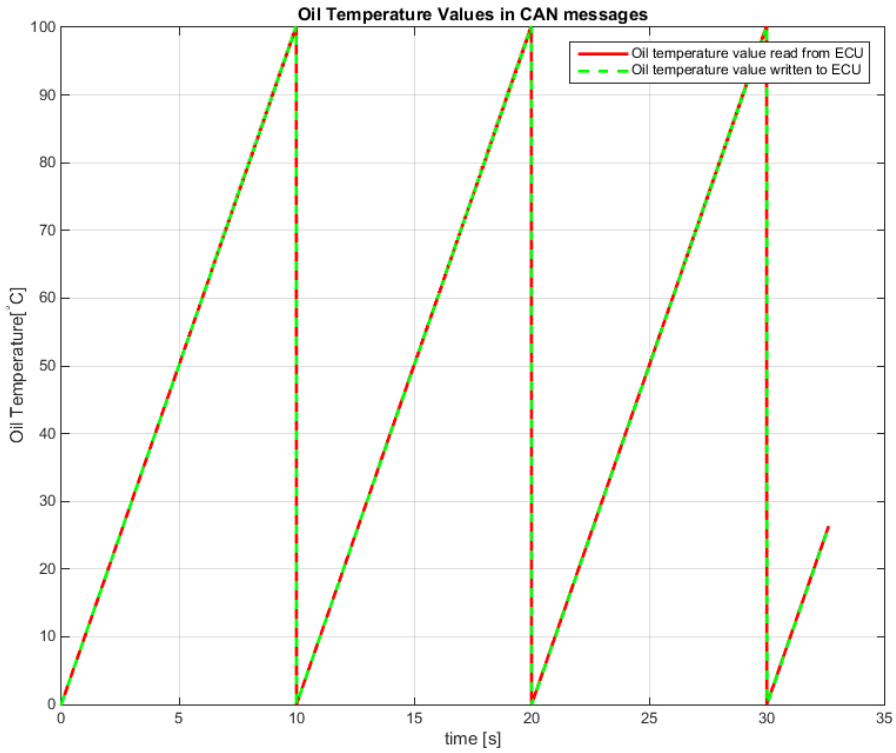

Figure 4.6: Comparison of the Oil temperatures sent and received via CAN during Polling

The figure shows that the final function output value and the ECU-calculated

duty cycle value remains the same until the point where calibration is enabled. At this point, the final function output obtains the RPS-calibrated duty cycle value again, and this increases to a 1.1 times bigger value than the ECU-calculated one.

The CAN message flow during this measurement is shown in table .2. The data gathering process of 12 signals with polling takes 2.7 ms. The calibration process is the same as with the previous measurement, and it lasts again a total of 0.8 ms. In this case too, the oil temperature values are extracted from the CAN message log and plotted in figure 4.6. As evident from the figure, the two signals, in this case, are also overlapping each other.

The communication speed between the RPS and target ECU sets the limitation of this implementation. For proper usage of this RPS implementation, one must consider how many signals need to be read, how long it takes for the RPS to execute the function to be bypassed and how many signals need to be calibrated. Thereby DAQ provides much faster measurements, with 12 signals being measured within 0.9 ms, while with polling, it took 2.7 ms to measure the same signals. The measurement, bypass function execution, and calibration must all be performed within the cycle time of the function to be bypassed.

4.2.3 Intelligent Watchdog

Figure 4.7 shows the signal flows during the measurement.

The watchdog is configured to act as a redundant system for the duty cycle function and hence executes the same function as the ECU. This way, the watchdog evaluates the proper execution of the function in the ECU. Figure 4.7 shows the measurement results of the calculated (red) and the monitored (green) duty cycle in the upper plot. The lower plot shows the watchdog counter variable (blue), the watchdog trigger variable (red), and the variable for simulating the fault (orange).

The fault is simulated by setting the stop_transfer variable. When the stop_transfer variable is set, the ECU-obtained duty cycle is held at a constant

value, which indicates a fault such as, for example, break of communication.

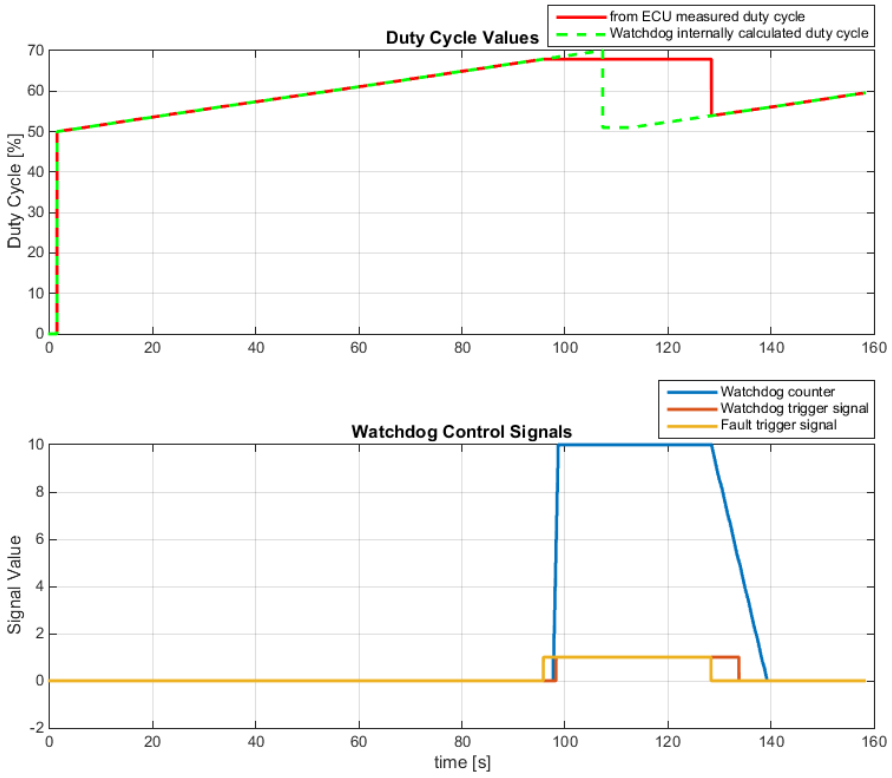

Figure 4.7: Watchdog signal monitoring

This leads to an increased deviation between the two duty cycle values, and when it reaches a predefined limit, the watchdog counter begins to rise. When the counter reaches a limit value (in this case, the value of 10), the watchdog trigger variable gets set and thus triggers the watchdog fault detection. Afterward, when the fault is disabled and the duty cycles are of the same value again, the watchdog counter begins to decrease. After it reaches a limit value again, it indicates that the fault is not present anymore, and the watchdog trigger gets deactivated. This represents the watchdog recovery stage.

For watchdog purposes, only DAQ transfer mode is viable since it obtains data within the same computation cycle.

4.2.4 Ethernet-CAN Interface

The busload measurements for 4-byte signals and 1-byte signals are shown in figures 4.8 and 4.9, respectively.

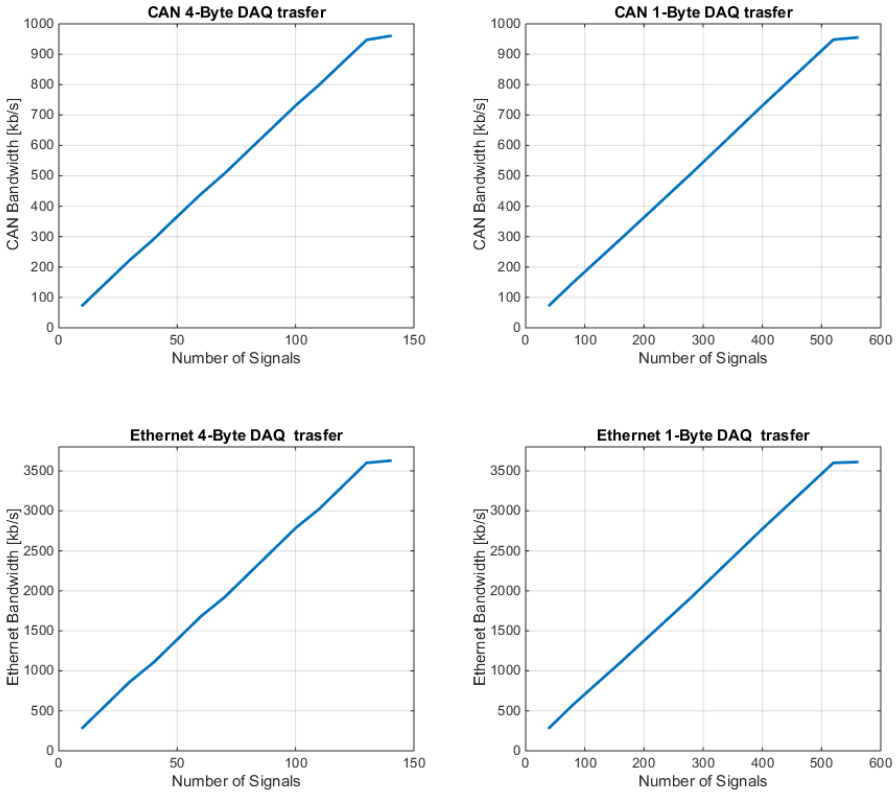

Figure 4.8: CAN and Ethernet Bandwidth measurements with DAQ

The implementation of the UDP and CAN transport layers make it possible to exchange the data from one bus to another easily. For testing the limits of

this implementation, bus load measurements are performed and evaluated. The baud rate, hence the maximum bandwidth of the CAN bus is 1 Mb/s.

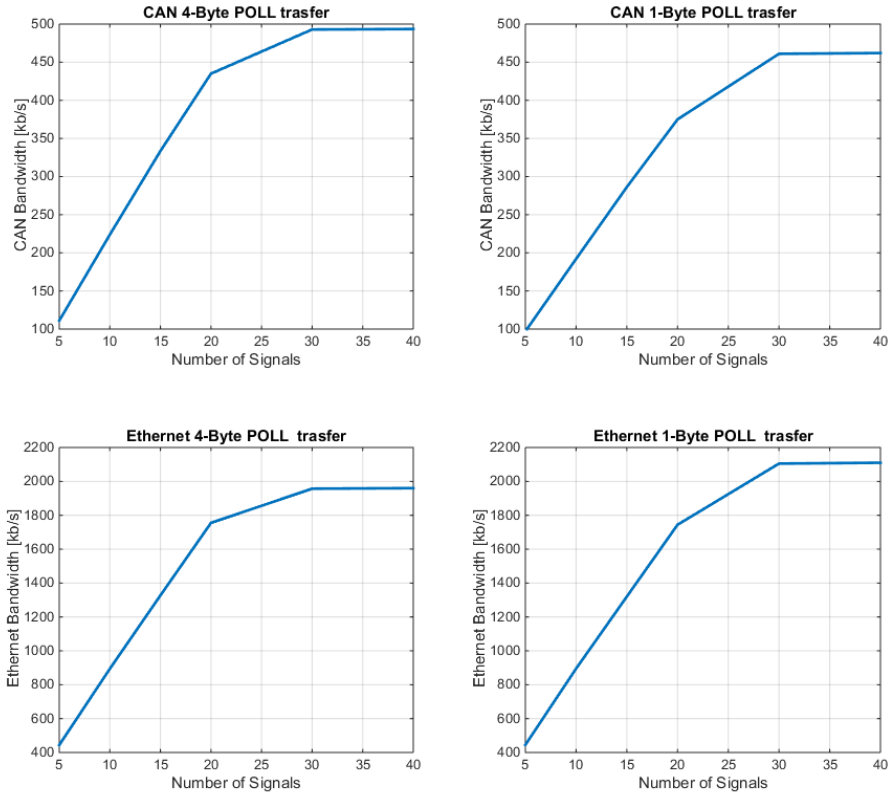

Figure 4.9: CAN and Ethernet Bandwidth measurements with Polling

Figure 4.8 shows the measurements obtained during a DAQ transfer of 4-byte and 1-byte signals. The x-axis represents the number of signals transferred over the bus while the y-axis represents the bandwidth caused by those signals in kb/s. As can be observed, the bus bandwidth rises linearly with the number of signals. The left-hand side of the figure shows the bandwidth of the 4-byte signals. It can be observed that the CAN bus load reaches its maximum when 140 signals are configured for transmission. The Ethernet bus load is higher than the CAN bus load because Ethernet

messages have much higher overhead than CAN messages. The right-hand side of the figure shows the bandwidth of the 1-byte signals. It can be observed that the maximum CAN bus load is now caused by 560 signals, which is exactly four times higher compared to the 4-byte signals. This behavior is ensured by the definition of the DAQ transfer as described in chapter 2.4.6. DAQ lists have 4 data bytes available for transferring in one message. In this case, CANape configured the DAQ transfer to send four 1-byte signals with one DAQ message, while in the previous configuration, it could only be sent one 4-byte signal per message. Hence, with the same amount of CAN messages it is possible to measure four times more 1-byte signals with the DAQ configuration compared to 4-byte signals.

Figure 4.9 shows the measurements obtained during data transfer with Polling of 4-byte and 1-byte signals. The left-hand side of the figure shows that the CAN bus load reaches its maximum value at 30 signals now. Furthermore, the maximum CAN bus bandwidth is around 500 kb/s now. Since polling requires a message from XCP-Master for every measurement signal, it causes a much higher bus load compared to DAQ, as described in chapters 2.4.5. and 2.4.6. The average time between two poll requests from master to slave is 0.5 ms. Hence, in a 10 ms interval, only 20 signals could be reliably measured with polling. This idle time between two poll requests causes the bus bandwidth not to be used at its maximum capacity of 1 Mb/s. The same is also true for the 1-byte signals. With polling, a poll request is sent for every signal, regardless of the signal size. Hence, the same message traffic is required for 1-byte signals as well as for 4-byte signals. However, it can be observed that the maximum CAN busload with 1-byte signals is slightly smaller compared to the 4-byte signals. This is due to the fact that the message from slave to master, which contains the signal value, is smaller in size when only 1 data byte is sent back. Hence, three fewer bytes are sent back from slave to master in case of measurement of a 1-byte signal compared to the measurement of 4-byte signals.

The maximum busload of the CAN bus is the main limitation of this implementation. Furthermore, the chosen measurement transfer mode also impacts the performance significantly. With DAQ mode and 10 ms cycle time, 140 4-byte signals or 560 1-byte signals could be successfully transferred, while with Polling mode, only 20 signals could be successfully transferred.

5 Conclusion and Recommendations for Future Work

5.1 Conclusion

The evaluation of the implementations described in chapter 3 has shown that the XCP protocol provides very good performance and a good basis for performing measurements and calibration for bypassing purposes. The RPS implementation in this thesis provides satisfactory results, in line with the expectations. Since it is based on real ECU hardware, it gives insight into how the tested functions perform hardware-near to a real ECU. This can show problems with resource limitations early on.

The performance of this RPS implementation is mainly limited by the CAN communication. The larger the number of signals which need to be calibrated and measured, the more time is necessary for transferring CAN messages. The total time required for transferring CAN messages needs to be shorter than the bypass function cycle time so that the RPS can perform the bypass in a timely manner. The time required for a CAN message transfer depends mainly on the chosen method of data gathering. With DAQ, this time is minimized since the test ECU sends all signals by itself and minimizes the busload. Polling requires commands from the RPS for every signal and increases the busload and time needed for message transfer. Therefore, DAQ should always be used as the data gathering method, if possible. If the test ECU should have no DAQ transfer enabled, then polling should be used. This implementation has shown to perform well for 10 ms functions.

Furthermore, the performance of this implementation also depends on the performance of the target ECU. Since the RPS is waiting for a response from the target ECU every time it sends a message, a delay from the ECU side also causes a delay of the RPS, and this can thus lead to a cycle being skipped.

XCP also proves to be useful for watchdog purposes. Utilizing the DAQ measurement method to monitor variables in a target ECU shows to be very reliable. For watchdog functionality, only DAQ transfer is viable since it ensures data consistency. All signals sent via DAQ are guaranteed to be from the same computation cycle of the ECU. With polling, this would not be the case, and the values received by the watchdog would be inconsistent, which could lead to wrong conclusions.

The Ethernet-to-CAN interface is also performing according to the expectations. Since CAN has eight useful bytes per message, Ethernet messages must also be limited to 8 data bytes per message for this implementation. Ethernet has a much higher bandwidth than CAN and, therefore, this implementation is mainly limited by the maximum busload of the CAN bus. The measurement transfer mode has a significant impact on the performance of the interface. Hence, when DAQ with a cycle time of 10 ms is configured, the maximum number of signals which can be transferred is 140 for 4-byte signals and 560 for 1-byte signals. When Polling is used as the measurement method, the signal size has a slight impact on the busload, but the number of signals to be reliably transferred is limited to 20 signals.

5.2 Future Work

As the foundation of an XCP-Master implementation on a standard microcontroller platform with a basic functionality has been laid with this thesis, further extension of the supported XCP-Master commands would lead to increased usability of the platform. For example, with all necessary XCP-Commands implemented in the XCP-Master Controller, on-board flashing of ECUs in a vehicle could be performed.

Furthermore, the performance of the RPS functionality could further be improved by utilizing the STIM XCP-Command for Stimulation. This would enable event-based calibration and thus speed up and improve the calibration process.

Since Ethernet is the most future-proof communication bus for vehicles, extending the transport layer of the XCP-Master Controller to also support measurement and calibration over Ethernet would be a reasonable next step.

The development of an RPS configuration tool that would be able to connect to the platform over XCP on Ethernet and enable the configuration of the RPS parameters during runtime would also be a useful extension. This would make commercial XCP-Master applications such as CANape not needed.

Appendix

Time [ms]	CAN-ID	Message
10763.6	061A	01 00
10763.6	061A	02 00
10763.7	061A	03 00
10763.8	061A	04 01
10763.8	061A	05 00
10763.9	061A	06 00
10764.0	061A	07 00 00 00 00
10764.1	061A	08 00 00 20 C2
10764.2	061A	09 02
10764.3	061A	0A CD FF CC 3D
10764.4	061A	0B 00 00 00 00
10764.9	610	F6 00 00 00 90 11 34 80
10765.0	061A	FF
10765.1	610	F0 04 9A FF 99 3E 00 00
10765.3	061A	FF
10765.4	610	F6 00 00 00 08 10 34 80
10765.5	061A	FF
10765.6	610	F0 04 8D FF 5C 42 00 00
10765.7	061A	FF

Table .1: CAN messages during one measurement and calibration cycle using Polling

Time [ms]	CAN-ID	Message
18495.3	0610	F4 01 00 00 F7 46 00 70
18495.4	061A	FF 00
18495.5	0610	F4 01 00 00 EB 46 00 70
18495.6	061A	FF 00
18495.8	0610	F4 01 00 00 2F 4C 00 70
18495.8	061A	FF 00
18496.0	0610	F4 01 00 00 2E 4C 00 70
18496.1	061A	FF 00
18496.2	0610	F4 01 00 00 9A 7F 35 80
18496.3	061A	FF 01
18496.4	0610	F4 01 00 00 33 4B 00 70
18496.5	061A	FF 00
18496.6	0610	F4 01 00 00 37 4C 00 70
18496.7	061A	FF 00
18496.9	0610	F4 04 00 00 A8 14 00 70
18497.0	061A	FF 00 00 00 00
18497.1	0610	F4 04 00 00 98 14 00 70
18497.3	061A	FF 00 00 20 C2
18497.4	0610	F4 01 00 00 F8 4B 00 70
18497.5	061A	FF 02
18497.6	0610	F4 04 00 00 64 20 00 70
18497.7	061A	FF CD FF CC 3D
18497.9	0610	F4 04 00 00 98 23 00 70
18498.0	061A	FF 00 00 00 00
18501.4	0610	F6 00 00 00 90 11 34 80
18501.5	061A	FF
18501.6	0610	F0 04 CD FF 4C 3E 00 00
18501.7	061A	FF
18501.8	0610	F6 00 00 00 08 10 34 80
18501.9	061A	FF
18502.0	0610	F0 04 8D FF 5C 42 00 00
18502.1	061A	FF

Table .2: CAN messages during one measurement and calibration cycle using Polling

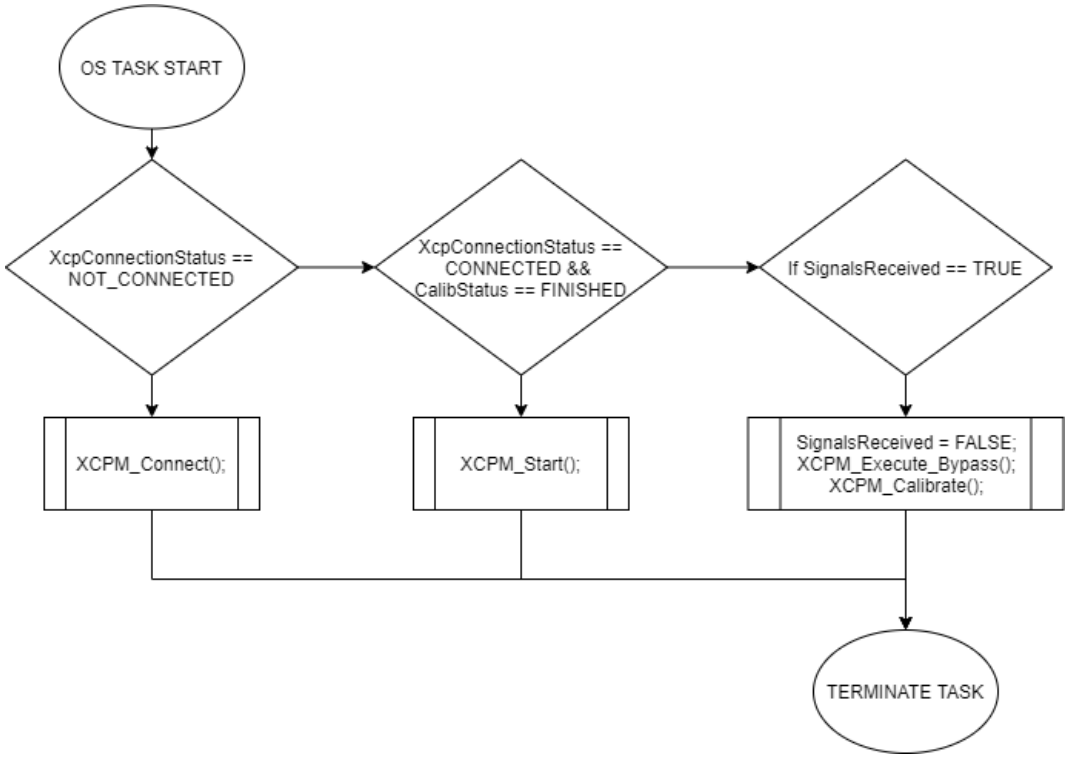

Figure .1: OS Task loop flow chart

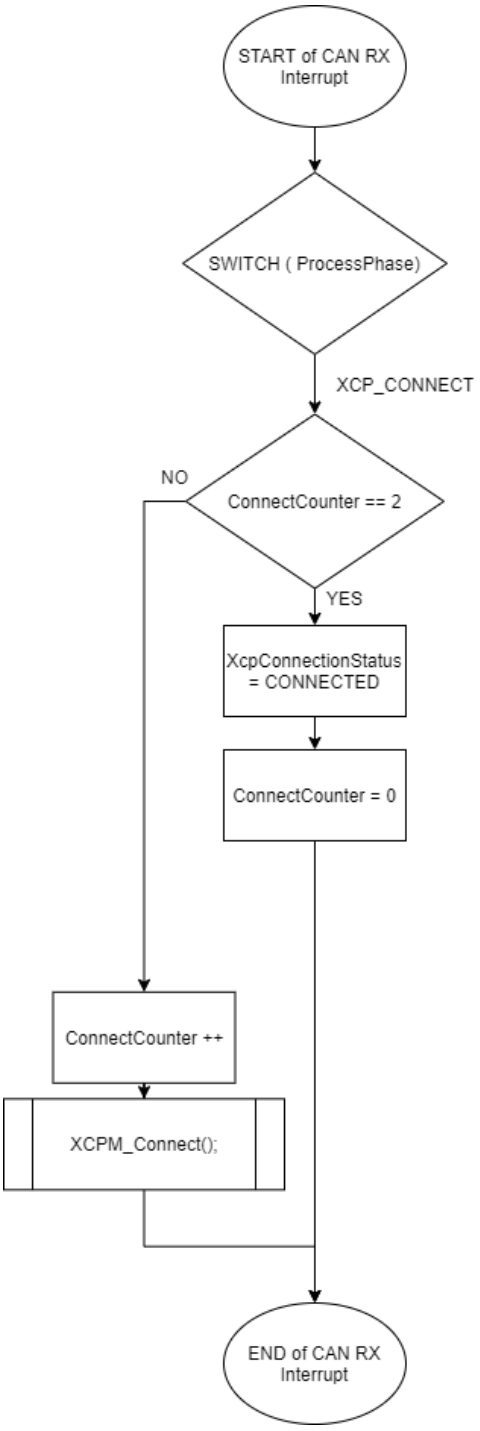

Figure .2: Connection process flow chart

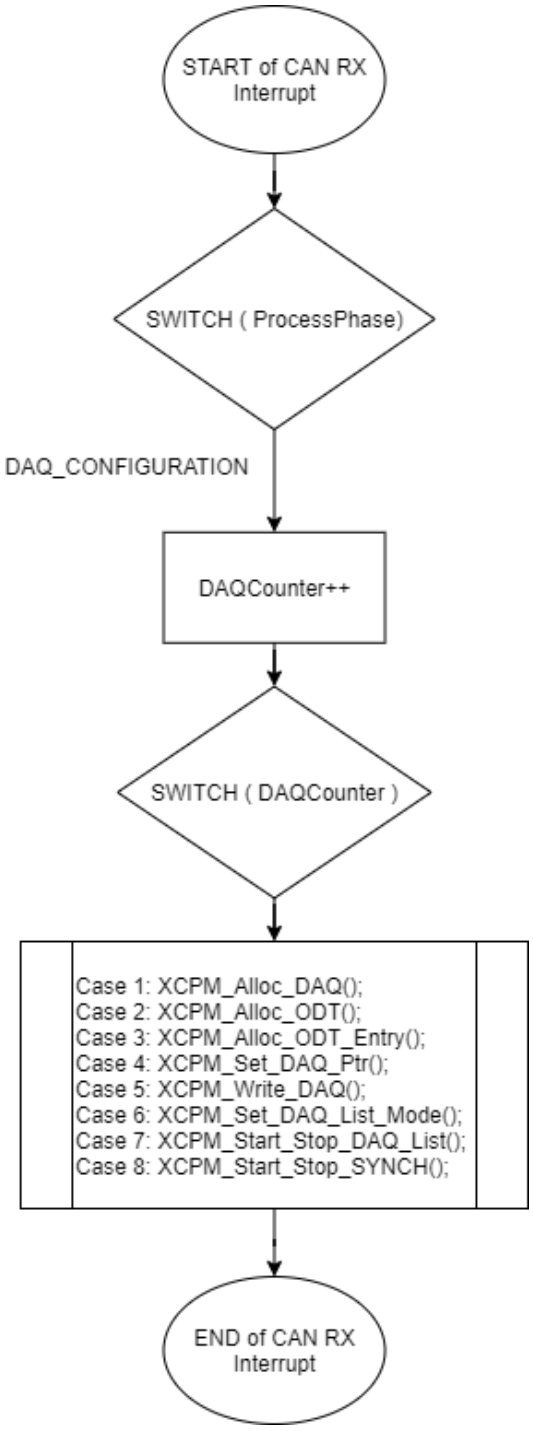

Figure .3: DAQ configuration process flow chart

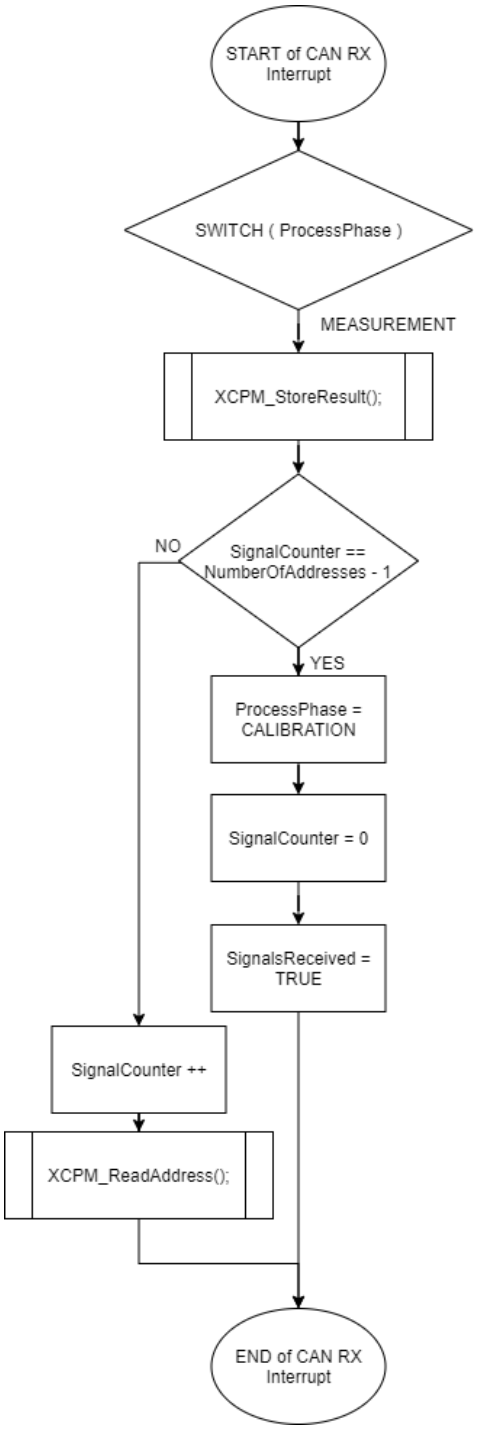

Figure .4: Measurement process flow chart

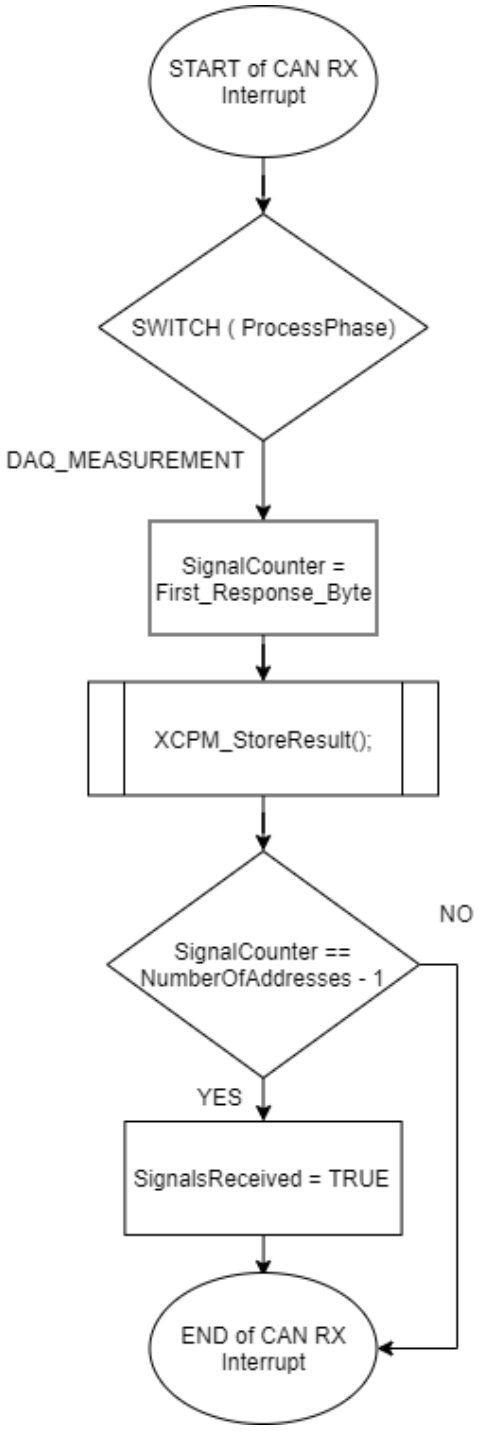

Figure .5: DAQ Measurement process flow chart

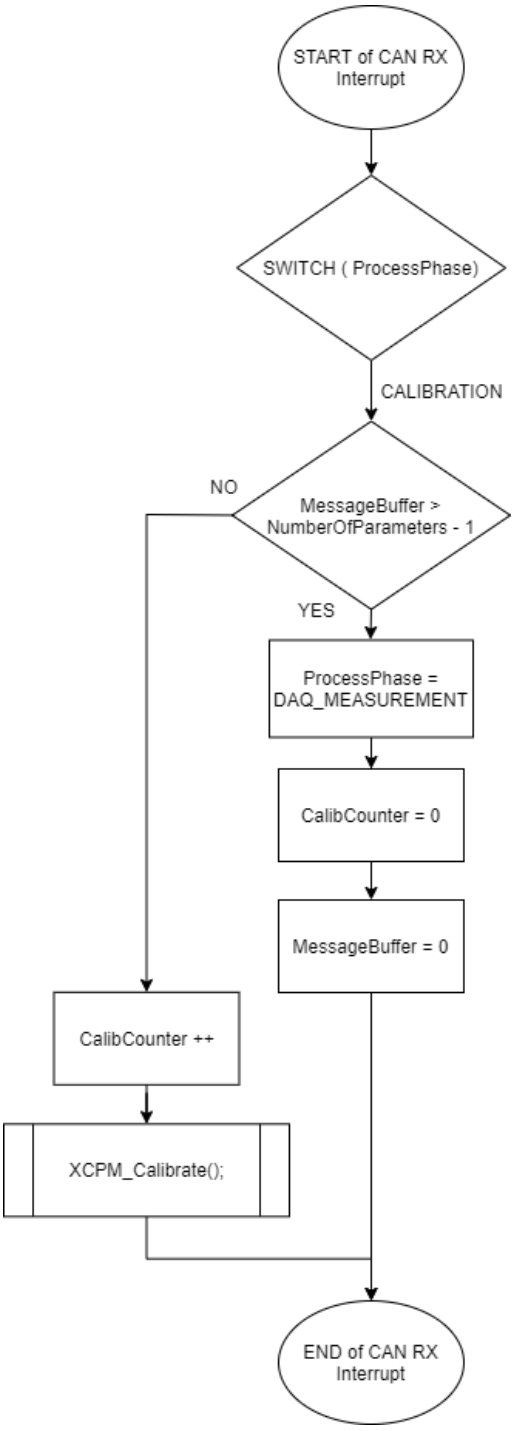

Figure .6: Calibration process flow chart

Bibliography

[1] Mohammed Alani. "OSI model." In: Mar. 2014, pp. 5–17. ISBN: 978-3-319-05151-2. DOI: 10.1007/978-3-319-05152-9_2 (cit. on p. 17).

[2] Patzer Andreas and Zaiser Rainer. "XCP - The Standard Protocol for ECU Development." In: (2016) (cit. on pp. 23–34).

[3] ASAM. "ASAM MCD-1 (CCP) CAN Calibration Protocol." In: (1999) (cit. on p. 21).

[4] ASAM. "ASAM MCD-1 (XCP on CAN) Universal Measurement and Calibration Protocol - CAN Transport Layer." In: (2017) (cit. on p. 24).

[5] ASAM. "ASAM MCD-1 (XCP on Ethernet) Universal Measurement and Calibration Protocol - Ethernet Transport Layer." In: (2017) (cit. on p. 25).

[6] ASAM. "ASAM MCD-1 (XCP) Universal Measurement and Calibration Protocol - Protocol Layer Specification." In: (2017) (cit. on pp. 21, 23, 27, 31).

[7] Alexander Barkalov, Larysa Titarenko, and Małgorzata Mazurkiewicz. "Introduction into Embedded Systems." In: *Foundations of Embedded Systems*. Cham: Springer International Publishing, 2019, pp. 1–22. ISBN: 978-3-030-11961-4. DOI: 10.1007/978-3-030-11961-4_1. URL: https://doi.org/10.1007/978-3-030-11961-4_1 (cit. on pp. 5, 6).

[8] Steve Corrigan. "Introduction to the Controller Area Network (CAN)." In: (2002/2016) (cit. on p. 15).

[9] Yanja Dajsuren and Mark van den Brand. "Automotive Software Engineering: Past, Present, and Future." In: *Automotive Systems and Software Engineering: State of the Art and Future Trends*. Ed. by Yanja Dajsuren and Mark van den Brand. Cham: Springer International Publishing, 2019, pp. 3–8. ISBN: 978-3-030-12157-0. DOI: 10.1007/978-3-030-12157-0_1. URL: https://doi.org/10.1007/978-3-030-12157-0_1 (cit. on pp. 6, 9).

[10] Deloitte. "Semiconductors – the Next Wave." In: (2019) (cit. on p. 6).

[11] dSpace. "Additional ECU Interface Solutions." In: (2020). DOI: https://www.dspace.com/shared/data/pdf/2020/dSPACE-Additional-ECU-Interface-Solutions_Product-Information_2020-01_EN.pdf (cit. on p. 22).

[12] dSpace. "MicroAutoBox II Brochure." In: (2020). DOI: https://www.dspace.com/shared/data/pdf/2020/dSPACE-MicroAutoBoxII_Product-Brochure_2020-02_EN.pdf (cit. on pp. 11, 22, 43).

[13] A. Dunkels. *Lightweight IP stack2.0.2.2017*. URL: https://www.nongnu.org/lwip/2_1_x/group__callbackstyle__api.html (cit. on p. 45).

[14] Martin Eckmann and Frank Mertens. "Close-to-ptoduction prototyping." In: *ATZelectronics Worldwide* (2006). DOI: 10.1007/BF03242303 (cit. on p. 11).

[15] ETAS. "ES910 Prototyping and Interfacing Module." In: (2018). DOI: https://www.etas.com/download-center-files/products_ES900/ES910_05-18_EN.pdf (cit. on pp. 22, 43).

[16] ETAS. "FETK User Interfaces Brochure." In: (2018). DOI: https://www.etas.com/download-center-files/products_ETK/FETK_05-18_EN.pdf (cit. on p. 22).

[17] Peter Hank, Thomas Suermann, and Steffen Müller. "Automotive Ethernet, a Holistic Approach for a Next Generation In-Vehicle Networking Standard." In: *Advanced Microsystems for Automotive Applications 2012*. Ed. by Gereon Meyer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–89. ISBN: 978-3-642-29673-4 (cit. on pp. 12, 17).

[18] Rajeshwari Hegde, Geetishree Mishra, and Gurumurthy Kargal. "Software and Hardware Design Challenges in Automotive Embedded System." In: *International Journal of VLSI Design Communication Systems* 2 (Sept. 2011). DOI: 10.5121/vlsic.2011.2314 (cit. on p. 6).

[19] Gerry Howser. "The OSI Seven Layer Model." In: *Computer Networks and the Internet: A Hands-On Approach*. Cham: Springer International Publishing, 2020, pp. 7–32. ISBN: 978-3-030-34496-2. DOI: 10.1007/978-3-030-34496-2_2. URL: https://doi.org/10.1007/978-3-030-34496-2_2 (cit. on p. 18).

[20] Beningo Jacob. "A Review of Watchdog Architectures and their Application." In: (Apr. 2010) (cit. on p. 35).

[21] Georg Macher et al. "Safety and Security Aspects of Fail-Operational Urban Surround perceptION (FUSION)." In: Oct. 2019, pp. 286–300. ISBN: 978-3-030-32871-9. DOI: 10.1007/978-3-030-32872-6_19 (cit. on p. 35).

[22] Alexander Malinowski and Bogdan M. Wilamowski. "User Datagram Protocol - UDP." In: *Industrial Communication Systems*. Boca Raton: CRC Press, 2019. DOI: https://doi.org/10.1201/9781315218434. URL: https://doi.org/10.1201/9781315218434 (cit. on pp. 19, 20).

[23] S. et al. Otten. "Automated Assessment and Evaluation of Digital Test Drives." In: *Zachäus C., Müller B., Meyer G. (eds) Advanced Microsystems for Automotive Applications 2017*. Cham: Springer, 2017. ISBN: 978-3-319-66972-4. DOI: https://doi.org/10.1007/978-3-319-66972-4_16 (cit. on p. 10).

[24] A. Rayes and S. Salam. "The Internet in IoT—OSI, TCP/IP, IPv4, IPv6 and Internet Routing." In: *Internet of Things From Hype to Reality*. Cham: Springer, 2017. ISBN: 978-3-319-44860-2. DOI: https://doi.org/10.1007/978-3-319-44860-2_2 (cit. on pp. 18, 19).

[25] "Automotive Software Engineering: Past, Present, and Future." In: *Automotive Mechatronics*. Ed. by Konrad Reif. Wiesbaden: Springer Vieweg. ISBN: 978-3-658-03975-2. DOI: https://doi.org/10.1007/978-3-658-03975-2. URL: https://doi.org/10.1007/978-3-658-03975-2 (cit. on pp. 7, 8, 12–15).

[26] Hans-Christian Reuss et al. “Efficient Automatic Application in Rapid Prototyping Software Development.” In: *ATZelectronics Worldwide* (2019). DOI: 10.1007/s38314-019-0059-8 (cit. on p. 11).

[27] F. Triem. “XCP Protocol Layer, Technical Reference.” In: (2005) (cit. on pp. 44, 46).

[28] Vector. “VX1000 Product Information.” In: (2018). DOI: https://assets.vector.com/cms/content/products/vx1000/Docs/VX1000_ProductInformation_EN.pdf (cit. on p. 22).

[29] Michael Marvin Wolf. “Master Thesis: An open source XCP based measurement and calibration system for automotive ECUs.” In: (2018) (cit. on pp. 45, 46).