CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Control Engineering

Master's Thesis

Diagnostic tools for car control units

Bc. Jakub Jíra
Open informatics – Computer engeneering

January 2024
Supervisor: Ing. Michal Sojka, Ph.D.

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Jíra Jakub** Personal ID number: **483779**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Measurement**

Study program: **Open Informatics**

Specialisation: **Computer Engineering**

II. Master’s thesis details

Master’s thesis title in English:

Diagnostic tools for automotive ECUs

Master’s thesis title in Czech:

Diagnostické nástroje pro ídicí jednotky aut

Guidelines:

1. Familiarize yourself with the UDS and ISO-TP protocols. Learn about the existing tool called "diag".
2. Implement a library for the UDS protocol communication in the Rust programming language. The library should offer an async/await API.
3. Port the "diag" application for diagnosis of PDC/PLA electronic control units from C to Rust using the developed library.
4. Create a usable user interface for the application and enable command line functionality.
5. Extend the developed application to allow diagnostics of multiple control units by loading a so-called PDX file containing information about the control unit in question. Test the result on both PDC/PLA unit and on the TSI 1.5 engine control unit.
6. Publish the resulting application as an open-source project.

Bibliography / sources:

VW80124_EN.pdf
https://udsoncan.readthedocs.io/
https://crates.io/crates/socketcan-isotp

© VUT v Praze, Design: VUT v Praze, VIC

Name and workplace of master's thesis supervisor:

Ing. Michal Sojka, Ph.D. Embedded Systems CIIRC

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **29.06.2023** Deadline for master's thesis submission: **09.01.2024**

Assignment valid until:
by the end of winter semester 2024/2025

Ing. Michal Sojka, Ph.D. Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

. Date of assignment receipt	Student's signature

© VUT v Praze, Design: VUT v Praze, VIC

Acknowledgement / Declaration

I would like to express my sincere gratitude to my supervisor, Ing. Michal Sojka, Ph.D., for his invaluable guidance, support, feedback, and kindness throughout the completion of this thesis.

I would also like to thank my wonderful fiancee Anna for her unwavering support, understanding, and love.

Lastly, I would like to thank my family, Kittens, and other friends, for their support and shared beers and laughs.

I declare that the presented work was developed independently and that I have listed all sources of the information used within it in accordance with the Methodical Instructions for Observing the Ethical Principles in the Preparation of University Theses.

In Prague, January 9, 2024

. . . .

Abstrakt / Abstract

Vývoj automobilových aplikací vyžaduje přístup k diagnostickým údajům řídicích jednotek (ECU). Zatímco existuje mnoho komerčních aplikací, které tuto funkci poskytují, nabídka řešení s otevřeným zdrojovým kódem je mnohem menší a často má omezenou funkčnost.

Tato diplomová práce se tuto situaci pokouší zlepšit. Představuje návrh a implementaci tří knihoven potřebných pro vývoj diagnostické aplikace napsanou v programovacím jazyce Rust. Problémy, které knihovny řeší, jsou asynchronní komunikace přes protokol ISO-TP, asynchronní komunikace přes UDS a parsování databází popsaných standardem ODX.

Jsou uvedeny dvě ukázkové aplikace využívající vyvinuté knihovny. Aplikace pro příkazový řádek, která umožňuje provádět základní diagnostické služby a číst data uložená v připojené řídicí jednotce. A aplikace zobrazující proces převodu přijatých dat na reálné hodnoty.

Klíčová slova: Rust, asynchronní, ISO-TP, UDS, ODX

Development of automotive application requires access to diagnostic data of the engine control units (ECUs). While there are many commercial applications providing that functionality, the offer of an open-source solution is much smaller and often of limited functionality.

This diploma theses attempts to improve that. It presents the design and implementation of three libraries required for an automotive diagnostic application written in the Rust programming language. The problems solved by the libraries are asynchronous communication over ISO-TP protocol, asynchronous communication over UDS, and parsing of the ODX database.

Two example applications using the developed libraries are presented. The command line application that allows the execution of basic diagnostic services and the reading of the data stored in the connected ECU. And the application showing the process of translating the received data into real-world values.

Keywords: Rust, asynchronous, ISO-TP, UDS, ODX

/ Contents

1 Introduction **1**

2 Background **3**

2.1 Communication protocols 3
2.1.1 UDS 3
2.1.2 ISO-TP 7
2.1.3 CAN 8
2.2 Used programming technologies . 8
2.2.1 Rust 8
2.2.2 Async/Await 9
2.2.3 Asynchronous runtime 9
2.3 ODX 10
2.4 Current drivers and implementations 11
2.4.1 Linux driver 11
2.4.2 socketcan-isotp 11
2.4.3 tokio-socketcan 11
2.4.4 python-udsoncan 11
2.5 Custom vehicle diagnostic tools 12
2.5.1 OpenVehicleDiag 12
2.5.2 odxtools 12

3 Design **13**

3.1 Overview 13
3.2 Async Rust ISO-TP library . . 14
3.3 Async Rust UDS library 14
3.4 ODX Parser library 15
3.5 Application design 16

4 Implementation **18**

4.1 tokio-socketcan-isotp 18
4.2 uds-rs 20
4.3 odx-parser 22
4.3.1 Deserialization 23
4.3.2 Resolving ODX references . 24

5 Evaluation **27**

5.1 Testing the functionality of the libraries 27
5.2 Creating sample application . . 27
5.2.1 Basic CLI UDS application 27
5.2.2 Interpreting ECU response using an odx-parser 29

6 Conclusion **31**

6.1 Future work 31

References **32**

A Abbreviations **37**

Tables / Figures

2.1 UDS request 4
2.2 UDS positive response 4
2.3 UDS negative response 4
2.4 ReadDataByIdentifier request . . . 5
2.5 ReadDataByIdentifier positive response 5
2.6 ReadDTCInformation request . . . 5
2.7 ISO-TP Frame types 7

1.1 TSI 1.5 simulator 1
1.2 Bosch Parking ECU 2
2.1 UDS sessions 6
3.1 Application architecture 13
3.2 Application architecture with API calls 17
4.1 Folder structure of uds-rs 21
5.1 CLI Appplication help 28
5.2 CLI Appplication command help 28
5.3 CLI Appplication ECU response 29

Chapter 1
Introduction

In the world of the modern automotive industry, electronic control unit (ECU) technology is becoming more complex; as automakers strive to meet increasing demands on the safety standards, fuel consumption, emmitions, and enhances in the driving experience, this complexity is only going to increase. And the current trend of software-defined vehicles is not going to make the ECUs any simpler.

In this world of complex ECUs, diagnostic software is essential for the development and testing of these convoluted systems. Most of the software used today in the industry is proprietary, expensive, and cannot be easily modified.

The topic for this thesis arises from different work: Simulator of the TSI 1.5 combustion engine [1]. The proprietary tools for developing the simulator are sufficient but not ideal.

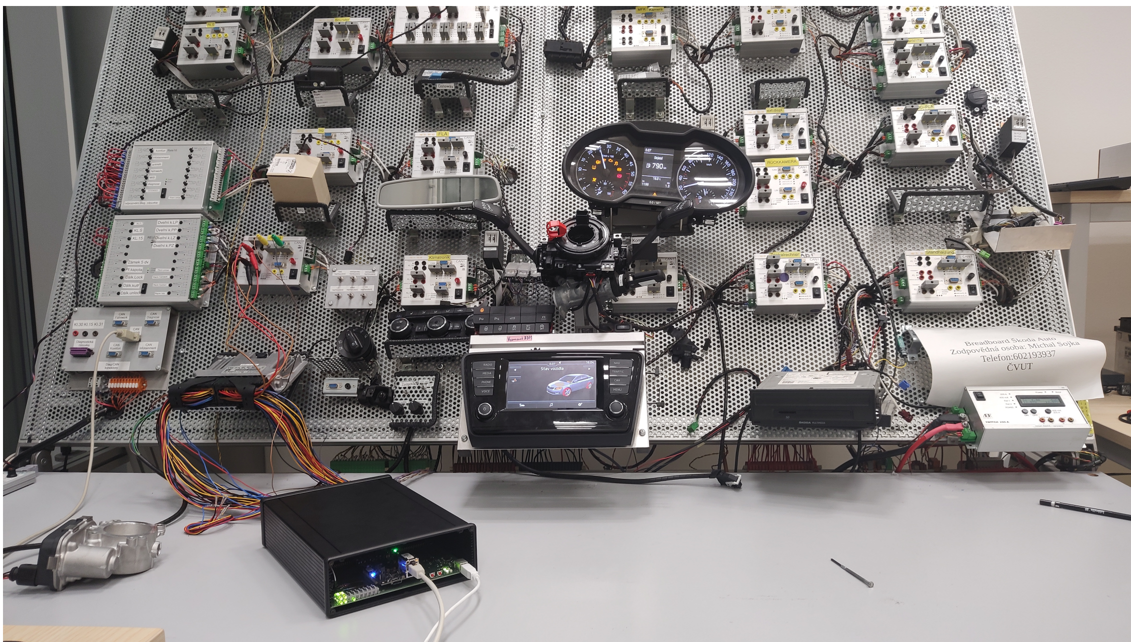

Figure 1.1. Photo of TSI 1.5 combustion engine simulator created by Jan Vojnar

This thesis aims to design and implement software libraries that would be required for a diagnostic application written in the Rust programming language.

Chapter 2 of this thesis covers the theoretical background of automotive communication protocols used for diagnostic, the Rust programming language and the async/await programming style, the Open Diagnostic data exchange format and current implementations of communication standards and open-source diagnostic applications. Chapter 3 describes the design of the libraries needed for a diagnostic application. Chapter 4 describes the implementation process of the libraries and the challenges encountered when

Figure 1.2. Photo of Bosch PDU Parking ECU, on which most of the tests were carried out

implementing them. Chapter 5 describes how tests were carried out and presents example applications. The last part of the thesis is Chapter 6 where the main results are described.

Chapter 2
Background

This chapter provides a comprehensive view of the technologies that are used by the project or have had an influence in another way. This chapter serves as a basis for subsequent chapters and provides readers with the necessary knowledge to understand the design and implementation of the project.

The first half of the chapter describes used communication protocols and standards. The second half goes through the current state-of-the-art and introduces current implementations of standards and even whole applications.

This chapter is not exhaustive and only covers topics relevant to the scope of the project. For more information, please refer to the cited standards and documentation.

2.1 Communication protocols

Current cars are large distributed systems full of various communication interfaces, ranging from simple passing of analog signals to fast, real-time networks. As the goal of this project is car diagnostics, this section of the document focuses on diagnostic communication stack.

After reading this section, the reader will be familiar with the communication protocols used by the project, namely UDS, ISO-TP, and CAN.

2.1.1 UDS

UDS or Unified Diagnostic Services, standardized by ISO in ISO 14229-1 [2], is a application layer protocol widely used in automotive to provide diagnostic communication between the ECU and a tester. This project is based on ISO 14229-1:2013, which is an older version of the standard, and the newer ISO 14229-1:2022 is the current version. The project is also designed to be compatible with VW80124 [3] – proprietary VolksWagen standard, which is mostly similar to the UDS.

In addition to the application layer, the UDS also standardizes session layer services in ISO 14299-2. Lower layers are standardized for each physical layer separately. UDS can work in any modern automotive communication stack. For example, ISO 14229-3 specifies the so-called UDSonCAN – UDS diagnostic over CAN. But UDS standard is also defined for FlexRey, Ethernet, LIN, and K-Line.

For this project, the diagnostic over CAN is the most relevant.

The UDS communication structure is based on client-server communication, where the diagnostic application is a client and the ECU is the server. The UDS defines diagnostic services that serve to test, monitor, or diagnose the server.

Typically, communication follows a request-response pattern, but this is not an absolute rule. Some services allow to suppress the positive response. There are also services in which communication is subscription-based.

The base structure of the UDS request is displayed in table 2.1. In all tables in the current section, n stands for the length of the message.

byte 0	bytes 1..n-1
SID	Parameters and/or Data

Table 2.1. UDS request

The service identification (SID) is a byte value that specifies the service, followed by additional data that are service-specific.

The request is sent to the server that responds with either positive or negative response.

The positive response has the same structure with the difference that the SID increases by 0x40 2.2.

byte 0	bytes 1..n-1
SID+0x40	Parameters and/or Data

Table 2.2. UDS positive response

Note that the *Parameters and/or Data* is different for request and response.

The negative response, shown in table 2.3 is sent when the request cannot be executed, when, for example, the service is not supported or the server is busy.

byte 0	byte 1	byte 2
0x79	Rejected SID	NRC

Table 2.3. UDS negative response

In the first byte, *0x79* represents the SID of the negative response. Second, what SID was rejected, and third negative response code (NRC). The NRC stores the information about the error that resulted in a negative response. The list of NRCs and their meaning can be found in Table A.1 in [2].

UDS introduces functional blocks into which services are divided. These are:

Diagnostis and Communication Management functional unit
Data Transmission functional unit
Stored Data Transmission functional unit
IntputOutput Control functional unit
Routine functional unit
Upload Download functional unit

For this project, the important function units are the Data Transmission and Stored Data Transmission functional units.

Data Transmission functional unit contains services that allow reading the ECU data, as well as writing based on an address or special identifier called the Data Identifier (DID). For this project, the most prominent service of this functional unit is the *ReadDataByIdentifier* service. The request of ReadDataByIdentifier is described in table 2.4

The 0x22 is the SID of the ReadDataByIdentifier service, and that is followed by at least one Data identifier. The Data identifier is a 16-bit value associated with a certain value stored in the ECU.

	Parameter
byte 0	SID = 0x22
byte 1	DID#1
byte 2	
:	:
byte n-2	DID#m
byte n-1	

Table 2.4. ReadDataByIdentifier request

	Parameter
byte 0	SID = 0x62
byte 1	DID#1
byte 2	
byte 3	Data#1
:	
byte (k-1)+3	
:	:
byte (n-1)-(l-1)-2	DID#m
byte (n-1)-(l-1)-1	
byte (n-1)-(l-1)	Data#m
:	
byte n-1	

Table 2.5. ReadDataByIdentifier positive response

The positive response for ReadDataByIdentifier is described in table 2.5. The k represents the length of Data#1, and l represents the length of Data#m.

The structure of the response is similar to the request, with the difference that each DID is followed by the corresponding Data.

Stored Data Transmission functional unit contains only two services. ClearDiagnosticInformation and ReadDTCInformation.

The ReadDTCInformation allows the client to read the Diagnostic Troube Code (DTC) information from the server. This service provides 27 functions.

The ReadDTCInformation is able to provide all these functionalities due to the usage of a sub-function byte. Usage is shown in the table 2.6. Each sub-function has its own specific byte defined in Table 269 in [2].

byte 0	byte 1	byte 2..n-1
0x19	sub-function	Parameters and/or Data

Table 2.6. ReadDTCInformation request

Some of the provided sub-functions are:

- 0x01 – Retrive the number of DTCs matching a client defined DTC status mask.
- 0x02 – Retrive the list of DTCs matching a client-defined DTC status mask.
- 0x04 – Retrieve DTCSnapshot data, containing specific data records associated with a DTC stored upon detection of system malfunction.
- 0x06 – Retrieve DTCExtendedData associated with a client-defined DTC and status mask combination out from the DTC memory. DTCExtendedData adds more data for the failed DTC.
- 0x0E – Retrieve the most recently failed DTC within the server.

For complete list refer to the Chapter 11.3 of [2].

An example of how the VW80124 standard differs from the ISO norm is in the definition of ReadDTCInformation. While the UDS provides 27 sub-functions, the VW80124 provides only 5. Sub-functions supported by both the VW80124 and ISO 14229-1 are the examples provided above.

Another feature that needs to be mentioned is the UDS Sessions. The current session defines what diagnostic services are allowed for the connected tester. The UDS defines four types of sessions:

- defaultSession – Enables the default diagnostic functions to the client. This session is active after the device is started.
- ProgrammingSession – Enables all diagnostic services required to support the memory programming of the server.
- extendedDiagnosticSession – Enables all diagnostic services required to support the adjustment of functions.
- safetySystemDiagnosticSession – Enables diagnostic services to support safety system related functions.

Vehicle manufacturers can decide which sessions they will support and if they will add some manufacturer-specific ones, the only exception being the default session. But they can also decide which services will be available while in the default session.

The session switching is visually described in 2.1

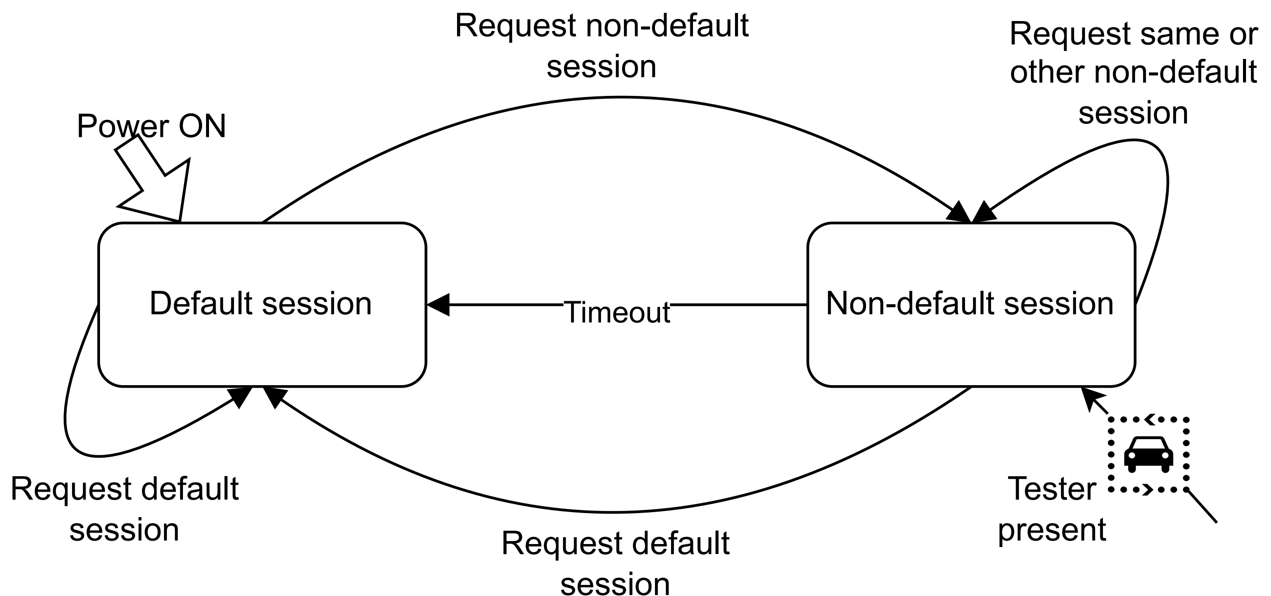

Figure 2.1. Diagram of UDS session switching

After the device is started, the default session is active. The tester can swich the current active session by using DiagnosticSessionControl service.

To remain in the non-default session, the tester needs to periodically send a TesterPresent request. If the tester fails to send the TesterPresent in time, the server returns to the Default session. The timeout is manufacturer-specific. The VW80124 standard defines this value as 5 000ms.

2.1.2 ISO-TP

ISO-TP (ISO 15765-2) [4] is a standard for transmitting data over a CAN bus. It was first published in 2004 by the International Organization for Standardization (ISO) as part of the ISO 15765 series of standards for vehicle networks. The standard was developed to address the growing need to transmit large amounts of data over a CAN bus.

ISO-TP is based on the existing CAN protocol. ISO-TP was designed to be compatible with existing CAN systems and provides a way to send messages longer than the 8-byte maximum payload size allowed by the standard CAN protocol. The maximum length of a message sent via the ISO-TP protocol is 4095 bytes. The 2016 implementation pushes this length to the theoretical maximum of 4GB.

From the perspective of the OSI model, ISO-TP serves the purpose of transport and network protocol layers.

While the CAN bus describes its messages as frames, ISO-TP messages are generally called packets.

Since ISO-TP is built on top of the CAN bus, the messages need to be parsed into CAN frames and sent fragmented. Also, the CAN bus is built without addressation in mind. Every message is broadcasted, and only the transmitter's address is known. ISO-TP allows six types of addressing. Addressing directly affects frame composition and payload size in each frame. One of these addressing methods is based on identifiers provided by the CAN bus, called normal addressing. In the following examples, normal addressing is used.

ISO-TP defines four types of CAN frames that are used to transmit the information. Those are *Single frame*, *First frame*, *Consecutive frame* and *Flow control frame*. The overview of these frames is shown in the following table.

Frame type	Byte 0		Byte 1	Byte 2	Byte 3..7
	Bits 7..4	Bits 3..0			
Single	Frame type	Payload size	Data		
First	Frame type	Payload size		Data	
Consecutive	Frame type	Index	Data		
Flow Control	Frame type	Flow status	Block size	Separation time	

Table 2.7. ISO-TP Frame types

Single frame is sent when the message is less than seven bytes.
- Frate type is 0.
- Payload size represents the number of data bytes.

First frame is sent when the message is more than seven bytes.
- Frame type is 1.
- Payload size represents number of Data bytes in current and consecutive frames.

Consecutive frame is sent after the First frame.
- Frame type is 2.
- Index is a four-bit counter, incremented with each sent frame.

Flow control frame is used by the receiver to inform the transmitter about the transmission of consecutive frames.
- Frame type is 3
- Flow status is 0 for continue, 1 for wait, and 2 for overflow/abort
- Block size represents the number of frames the transmitter is allowed to send before receiving another Flow control frame. If set to 0, the whole packet will be sent without awaiting another Flow control frame.
- Separation time specifies the minimal delay the transmitter needs to wait after sending any frame.

These four frame types are the basis for ISO-TP data transmission.

If the payload is smaller than seven bytes, a Single frame is used. After receiving a Single frame, the receiver does not send any Control flow frames.

In case of larger payloads, the transmitter will send the First frame and wait for the Flow control frame. After receiving Flow control frame, Consecutive frames are sent based on the received directives.

2.1.3 CAN

The controller area network (CAN bus) is a vehicle network designed by Bosch and later standardized by the ISO 11898 standard. CAN specifies the function of the physical and data-link layers [5]. The physical layer is described by ISO 11898-1, whereas ISO 11898-2 describes the physical layer. This project is based on CAN 2.0A and although there are newer standard CAN buses such as CAN-FD the CAN 2.0 is still the standard in the vehicle industry [6].

The basics of communication are that each message, called frame, has, among others, a unique ID and a data payload. The messages are broadcast to the entire network, and each node decides, based on the ID, whether to receive the message or ignore it. The ID also determines the priority of the message. During the transmission of the ID, the process called arbitration takes place.

Each node listens to the bus state while transmitting, and if the bus contains a different symbol than the one its transmitting, the transmition is stopped. This is possible due to the design of the physical layer of CAN, where the representation of bit 0 is dominant over the representation of bit 1.

The arbitration is won by the message with the lowest ID, and then the data is sent. This ensures that the highest priority message is transmitted first, without additional delay [5].

The main benefits of the CAN bus are low cost, high reliability, robustness, and data consistency across the network.

2.2 Used programming technologies

2.2.1 Rust

Rust is a programming language that was first developed at Mozilla Research in 2006 by Graydon Hoare. It was created to address the shortcomings of existing systems programming languages such as C and C++. Since its initial release, Rust has grown in popularity and is now a popular language for creating highly efficient, safe, and reliable

software. Rust enforces its safety with robust rules and mechanisms that prevent most of the behaviors that could end with a non-defined or not expected outcome [7].

This safety enforcement is carried out through a combination of multiple mechanisms, the main being ownership, borrowing, and lifetimes [8].

Ownership is a concept where each value has a variable that is its *owner*, and there can only be one owner at a time. When the owner goes out of the scope, the value is dropped. Ownership can be moved to another variable. This transfer is called *move*.

When we do not want to take the ownership, Rust allows us to borrow a value, creating a reference to the value, without transferring ownership. At one time, there can be either one mutable reference or multiple non-mutable ones.

These conditions of ownership and borrowing are checked during compile time by a core rust compiler functionality called the borrow checker. The lifetime of a reference is also checked during compile time, preventing the interaction with any out-of-scope data.

Rust also has a strong type system that helps prevent unintended data manipulation and type-related errors. Does not have garbage collection, but instead follows the RAII (resource acquisition is initialization) convention [9].

All these safety compliances are checked during compile time, which keeps Rust programs fast and safe simultaneously, making it suitable even for embedded applications.

Another valuable feature of Rust is its ecosystem. Apart from the compiler and standard library, Rust also consists of other utilities, all managed by rustup – a toolchain developed by the Rust project. The most notable is Cargo, Rust's build tool and package manager. Libraries or packages managed by Cargo are called crates and are typically distributed via crates.io, the official Rust package registry.

The newest version of Rust that this project was tested on is 1.74.1

2.2.2 Async/Await

Async/Await is a way to write code that allows for asynchronous execution but, from the perspective of the developer, looks and behaves as synchronous. This method was first introduced in C# in 2011 [10] and, since then, has been adopted by most modern concurrent programming languages. Rust programming language added support for async/await in 2019 in version 1.39 [11].

Async functions are unique in the way that they can be "paused", return the control to runtime, and then resumed at the same spot to continue the execution. For our implementation, the pause part is when our application waits for the I/O event on the ISO-TP socket.

Rust's approach to executing asynchronous routines is lazy [12]. When calling the async function, the future is created. Futures start execution only when the await or other function with a resolving effect is called on them. This approach allows users to easily separate the setup phase of creating asynchronous jobs and actually executing them.

To work properly, Futures require runtime that will correctly poll, schedule, and execute them. Rust does not provide native executors and reactors. However, the community-provided crates are designed to provide the desired functionality.

2.2.3 Asynchronous runtime

Asynchronous runtimes serve as libraries designed for the execution of asynchronous applications. Typically, these runtimes include a reactor paired with one or more executors. Reactors provide subscription mechanisms for external events, such as asyn-

chronous I/O, interprocess communication, and timers. Within an asynchronous runtime, subscribers commonly take the form of futures that represent low-level I/O operations. Executors manage the scheduling and execution of tasks. They are responsible for monitoring active and paused tasks, continuously polling futures until completion, and awakening tasks when there is an opportunity for progress [13].

As the Rust language evolved, several asynchronous runtimes were created. Currently the most prominent are Tokio [14], async_std [15] and smol [16].

The async_std is a runtime that tries to mimic as closely as possible the standard library of Rust while providing the async alternatives. Although the project is not “dead” [17], the developement is curently stagnating and majority of developers moved to smol or other projects.

The smol and async_std currently represent more niche runtimes. Smol focuses on a small and fast runtime independent from Mio(basis of Tokio), containing only safe code.

Tokio is by far the most widely used and is the de facto industry standard. It is based on Mio[18] (Metal IO), a low-level I/O library providing non-blocking APIs and event notifications. Tokio provides multi-threaded runtime with work-stealing scheduler, and asynchronous version of the standard library and a large ecosystem of libraries.

2.3 ODX

ODX is the abbreviation for Open Diagnostic data Exchange, and is standardized in ISO 22901 and ASAM MCD-2D. This project and all the information about ODX is based on ODX version 2.0.1.

The purpose of ODX is to provide a data format for the exchange of diagnostic specification of ECU diagnostic and programming data between the system suplier, the vehicle manufacturer, and service dealership [19].

The foundation of the ODX is the data model described in UML. Once the specification is completed, the data is mapped to the XML file.

The main blocks of the ODX database are:

- DIAG-LAYER-CONTAINER – Contains one or a set of DIAG-LAYER structures. These structures provide a description of diagnostic services with all the necessary data. The DIAG-LAYER structures are:
  - ECU-SHARED-DATA – provides a library-like mechanism, allowing data sharing across multiple ECUs.
  - PROTOCOL – generic services, e.g. UDS.
  - FUNCTIONAL-GROUP – generic diagnostic services of the same category
  - BASE-VARIANT – specific diagnostic services of specialized ECU
  - ECU-VARIANT – expanded or modified services of the BASE-VARIANT
- COMPARAM-SPEC – defines the standard parametrization of the communication protocols, e.g. timings.
- VEHICLE-INFO-SPEC – defines how can tester connect to the ECU.
- FLASH – definition of methods and parameters for flashing the ECU.
- MULTIPLE-ECU-JOB-SPEC – definition of diagnostic jobs that deal with quasi-parallel communication with several ECUs.

Each of the ODX files contains exactly one of the blocks.

The most relevant element in the above-mentioned list for this project is the DIAG-LAYER-CONTAINER. Here, the DIAG-SERVICE element specifies the REQUEST,

POS-RESPONSE, NEG-RESPONSE of certain service and DATA-DICTIONARY-SPEC contains all the needed data to create and parse the communication messages.

2.4 Current drivers and implementations

The described communication protocols and standards are widely used and there are numerous instances of existing implementations. This section introduces a few of these implementations closely related to the topic of the project.

2.4.1 Linux driver

SocketCAN is an open-source implementation of the CAN protocol stack for Linux. Provides a set of socket-based programming interfaces for working with CAN networks on Linux systems. SocketCAN enables developers to interact with CAN devices and networks using standard socket programming techniques, making it easier to integrate CAN communication into applications running on Linux platforms [20]

Since Linux kernel version 5.10 the ISO-TP communication is also part of the mainline SocketCAN driver [21]. The Linux community also developed SocketCAN-based utilities that are widely used and can serve as a basic diagnostic tool, when creating a CAN device [22]

SocketCAN is standard on all devices running Linux, ranging from embedded systems, automotive applications, and industrial automation. And is ideal to use as a base of this project.

2.4.2 socketcan-isotp

Socketcan-isotp [23] is a Rust library based on the mentioned Linux driver 2.4.1. The library API provides blocking I/O operations and a complete socket-creation process.

This library is created without the Async/Await communication in mind; however, this library allows the socket to be made unblocking, which is enough for using it as barebone communication and creating an asynchronous API on top of it.

2.4.3 tokio-socketcan

Tokio-socketcan[24] is a library based on Tokio runtime and adds an asynchronous layer on top of the socketcan-rs[25] crate. Since socketcan-rs version 3.0 the code from tokio-socketcan library was merged into the socketcan-rs and providing the Async/Await communication to not only tokio, but also async_std and smol.

Since both of those libraries only provide the CAN API and not ISO-TP these libraries are not suitable for our goal.

However, implementation of this library can give us a clear vision of how a Tokio library can be created on top of an existing one using AsyncFd, which will be helpful in our implementation.

2.4.4 python-udsoncan

The python-udsoncan [26] library is arguably the most feature-complete, open-source implementation of the UDS protocol. It is written in Python and provides most of the services through the API. Since the client is synchronous, it does not support asynchronous services such as ResponseOnEvent or ReadDataByPeriodicIdentifier.

Even though the code is unuseable by our Rust implementation, it can serve as an inspiration for the project.

2.5 Custom vehicle diagnostic tools

Diagnostic software working on the UDS communication protocol is not a new idea. There are various projects solving this problem, since our assignment specifies programming language, our options are far more limited.

Automotive software is heavily dominated by proprietary tools. This is true even for the diagnostic software. The most prominent is Vector Group, whose tools like CANoe, CANdelaStudio, or ODXStudio are the industry standard [27].

2.5.1 OpenVehicleDiag

The OpenVehicleDiag [28] is designed to diagnose cars without the need for expensive specialized software and hardware. As impressive as this software is, it is aimed mainly at enthusiasts to diagnose their cars and reverse engineers.

Also, the implementation of UDS lacks some crucial services like ReadDataByIdentifier.

The communication backend of this app is available as Rust crate [29], allowing it to be used in other projects. But even though the communication uses unblocking sockets, the whole communication process is done by passing the function to the thread and letting the thread do all the communicating. Since we are targeting acync/await communication, this would need to be reworked.

2.5.2 odxtools

The odxtools [30] is Python application by Mercedes-Benz that contains set of utilities to interact with the ODX data format.

The main functionality provided by odxtools is parsing and interpreting ODX diagnostic database files, as well as decoding and encoding the data of diagnostic requests and responses. It also provides API for basic communication with the ECU.

The main disadvantage is that the application only supports ODX 2.2.0. The implementation is also very memory inefficient, allowing only a small project to be loaded on my hardware[1].

[1] Valve Steam Deck running SteamOS 3.5.7

Chapter 3
Design

This chapter provides a high-level overview of the developed libraries and APIs, as well as the architecture of the final application.

3.1 Overview

The conclusion of the chapter 2 is that there are some publicly available libraries and applications that could be used as the basis for our diagnostic application.

The targeted application needs to be able to communicate with the UDS library through the Async/Await API, and needs to be able to read the data from ODX files, if the files are provided.

In order to write an Async/Await UDS library, it is required to have ISO-TP library, which would provide communication with the ECU through the Linux driver.

The visual representation is presented in the Figure 3.1

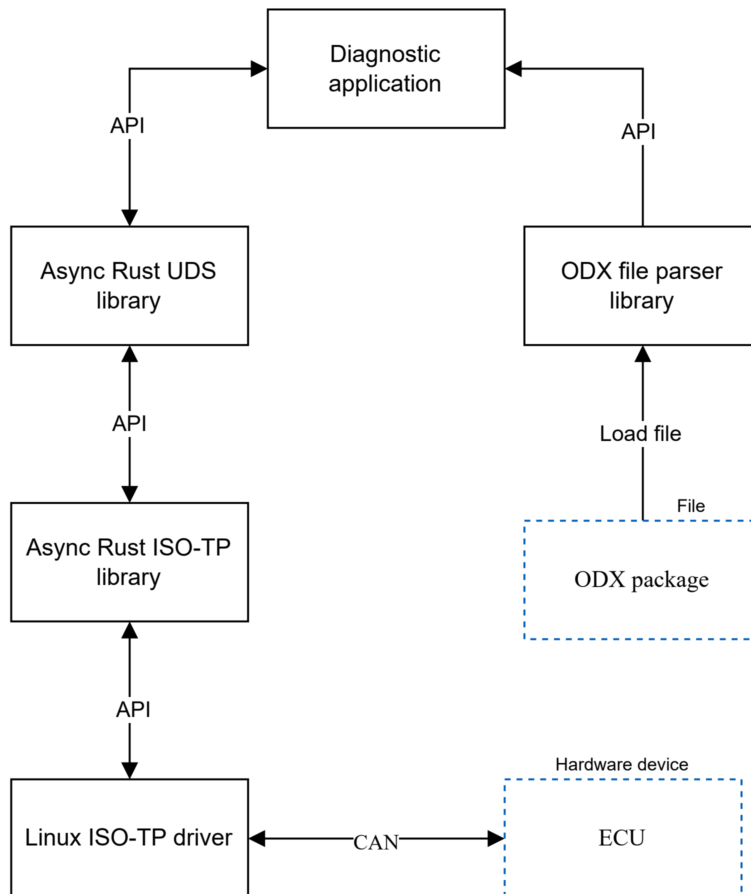

Figure 3.1. Application architecture

The Linux SocketCAN driver is ideal for low-level communication over CAN.

However, there is no Rust ISO-TP library that supports the async/await programming style. The existing library socketcan-isotp[23] is a suitable candidate to be the cornerstone of an asynchronous library.

The Async Rust UDS library is also missing and will need to be developed, as well as the ODX file parser library.

3.2 Async Rust ISO-TP library

The main function of the Async ISO-TP library is to provide wrapper around standard file descriptor and allow usage in the async context. I call this socket wrapper an AsyncSocket.

The creation of AsyncSocket should also provide all the settings for creating the socket as the socketcan-isotp library. Either by implementing the wrapper around existing functionalities, or transparently providing all the settings in the socketcan-isotp library.

The primary design choice involves an asynchronous runtime. That determines the features available which can help when designing and implementing the API that will be made available to the developer using the library.

As we discussed in the background section Asynchronous runtime 2.2.3, there is currently one major runtime environment for async rust – tokio [14]. Since we do not need any niche features, that the async_std or smol provide the tokio is a easy choice.

There are two standard approaches to designing an async API for the library.

One approach implements standard *write* and *read* methods for AsyncSocket which will return the type, which implements Future trait. Allows the calling method *poll* on that type, thus resolving the future.

Another approach is to use one of the traits provided by the futures-rs or tokio. The futures-rs provide stream and sink traits, hiding the futures behind this abstraction and providing *poll_next* and *start_send* methods, respectively[31–32]. The tokio provides AsyncRead and AsyncWrite traits[33–34].

Both approaches have their advantages and disadvantages; however, for this project, I have chosen to implement communication using the *read* and *write* methods, as it provides both greater control to the end user and to the developer for how to poll the Futures. The other means of interaction, as the traits mentioned, can be added in the future in the form of feature.

This approach is also independent of any third-party libraries, using only native Rust features. If the runtime should change in the future, the API could remain the same.

3.3 Async Rust UDS library

The main goal of the UDS layer is to provide basic UDS functionality through the async API. While allowing easy implementation of the new services and allowing for more complex UDS functions to be implemented in the future.

The UDS library should be independent on the back-end communication and should allow to receive UDS messages through other means, than iso-tp socket.

In addition to that, the UDS library should, if possible, be independent of the asynchronous runtime, utilizing whichever runtime the underlying communication layer is using.

The proposed architecture uses the UdsClient structure as the main source of API interaction, providing abstraction over the lower-level communication and providing methods for each implemented UDS service.

The UDS services can be implemented by creating a method for each service returning structure representing the corresponding expected response.

The UdsClient also provides more user settings, such as timeout for the message, or retry count, when the client receives a negative response – busy, try again.

To achieve independence on the background communication, the library should provide some abstraction over the used communication protocol, either through custom trait, or trough some wrapping layer.

Independence on runtime is achieved by using the asynchronous functionalities present in the Rust language, the *async* and *await* keywords, and the contents of module *std::future*. It is also possible to not use these functionalities at all and solve the asynchronous aspect only in the wrapping of the underlying layer, using bridging [35]. It would make it easier to use the library in a synchronous context, but since we specifically target the asynchronous Rust, the pure asynchronous approach is preferred.

3.4 ODX Parser library

One of the core functionalities of the application should be the ability to interpret received UDS data using the provided ODX file. To carry out the translation, several functionalities need to be implemented.

The basis is to provide some structure to which the XML code of the ODX file would be deserialized. Once parsed, the library should provide user with the entirety of the ODX file and let user decide what to do with parsed data.

One of the key features of the ODX format is that, in order to avoid repetition, certain sections of an ODX document can be referenced and, therefore, reused multiple times. ODX specifies two kinds or references. References via odx-link, and references via short-name.

The more complex of these two is the element ODXLINK which references the XML element by its attribute ID. This ID is unique in all ODX documents. The ODXLINK element contains attribure ID-REF, which contains the ID of the referenced element; odx-link provides three other optional attributes for cross-document references. When only the ID-REF attribute is provided, it signals that ID is stored in the current document.

The optional attributes DOCREF and DOCTYPE together specify the document in which the ID can be found. The documents to which can be referred in this way also follow a specified pattern. DOCREF specifies the SHORT-NAME of the target, and DOCTYPE specifies what the type of the target is.

The referenced documents can be one of the following:

1) Document specified in the IMPORT-REFS element.
2) Documents that are being inherited by the PARENT-REFS element.
3) Documents that inherit from the same direct parent as the current document.

All the references are recursive, so not only the direct parent is a possible target for reference, but also a parent's parent, and so on.

The point number 3 is especially problematic; since it is not possible to resolve the reference only from curent document and its direct references, one needs to parse the whole PDX folder in order to get the possible reference targets.

The other reference SNREF that references an XML element by its SHORT-NAME does not allow cross-document references and, therefore, is simpler. However, contrary to the ID, SHORT-NAME is not unique across the document but only in the specified boundary. These boudaries are element-specific (can be found in Table 12 in [36]).

There are many features in ODX, however, none is as prominent as the referencing. Other functionalities can be implemented on the per-app basis, but resolving ODX references needs to be as part of the basic library.

The basic API should take the ODX folder with specified ODX file as input and return the deserialized structure, as well as resolve all the references. Either statically importing them, or providing mechanism to lazily load them during run-time.

3.5 Application design

The libraries designed fill the missing pieces for the creation of diagnostic applications. An example of a rough usage can be found in Figure 3.2.

The figure expands upon the Figure 3.1 by not using generic names, but instead uses the names of the real implemented structures, methods and libraries.

Please note that the figure is **not** exhaustive and highly simplified. For complete documentation, refer to the Implementation 4 chapter, or the documentation of the corresponding modules.

Due to the simplification, return types are also simplified, omitting errors by not returning Results and Options which are in the implemented software.

The architecture consists of three libraries presented in the previous sections. The *tokio-socketcan-isotp* library is the wrapper around CAN communication with the ECU. Providing the strucutre *IsoTpSocket* with the *read_packet* and *write_packet* methods. The returned types of *IsoTpReadFuture* and *IsoTpWriteFuture* are custom types that implement the future trait.

The *UdsClient* in the *uds-rs* library uses the *IsoTpSocket* to communicate with the ECU and provide the end user with methods that correspond to UDS services. Due to the simplification of the figure, only two services are shown. Both of these services return an enum EcuResponse, which unifies all the possible responses.

The odx-parser loads the ODX package and deserializes its contents into the struct *ODX*. In order to keep the schema simple, the tree structure of odx is left out, and only *odx_element* represents its existence.

OdxStructure holds the *Odx* tree, as well as the HashMap of all imported and parent refs. Providing user with the contents of ODX package as well as methods like *resolve_odx_link(..)* that returns reference to the searched element.

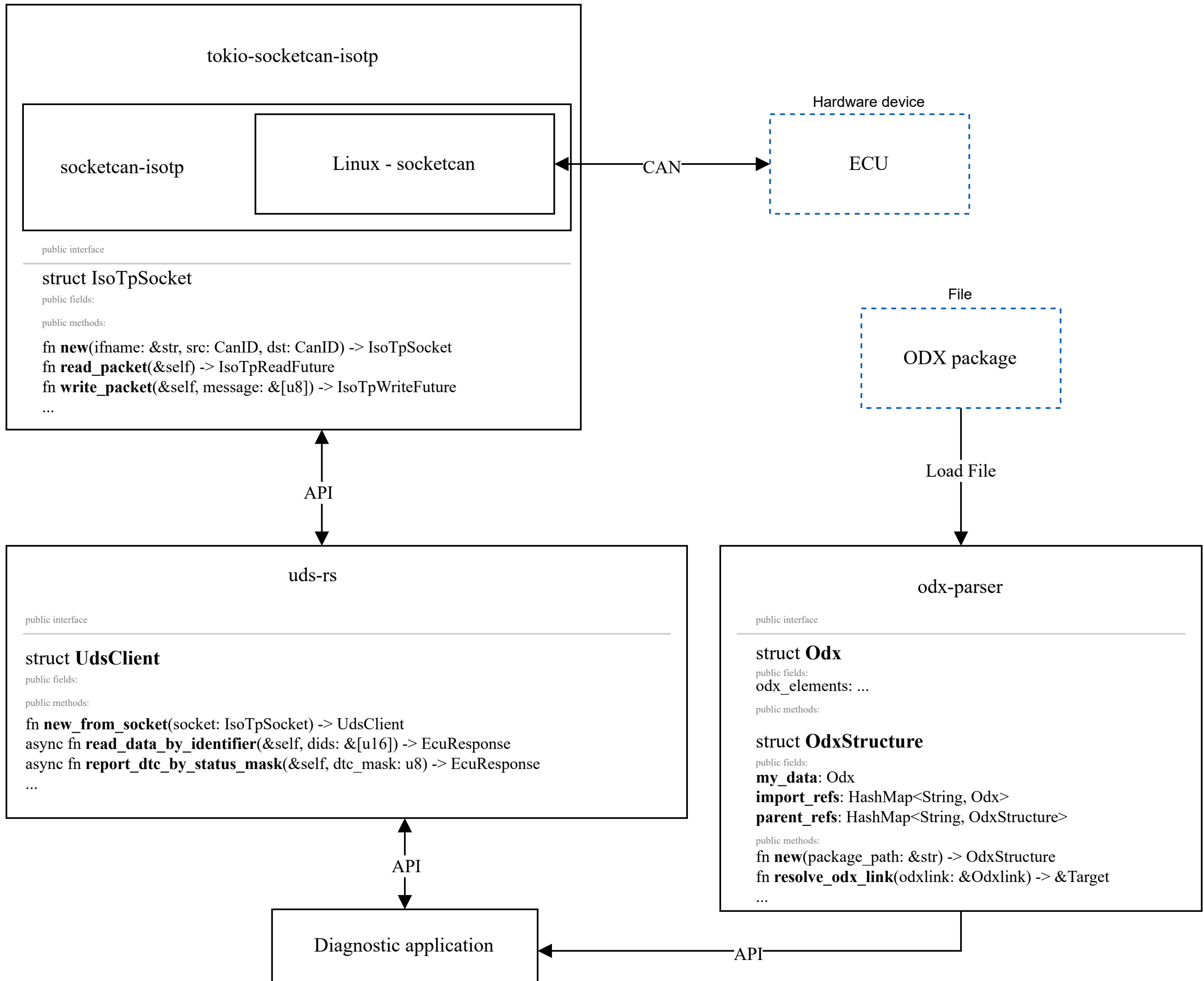

Figure 3.2. Application architecture with API calls

Chapter 4
Implementation

This chapter provides a deep dive into the implementation of libraries. Documents not only the implementation decisions and challenges, but could also be used as documentation for developers using the libraries.

4.1 tokio-socketcan-isotp

The code discussed in this section is freely available on crates.io [37].

The chosen runtime tokio provides the AsyncFd [38] feature which simplifies calling async operations on the unix file descriptor. Using this feature, the definition of the IsoTpSocket is following:

```
pub struct IsoTpSocket(AsyncFd<socketcan_isotp::IsotpSocket>);
```

Now we can call *poll_write_ready()* and *poll_read_ready()* on the file descriptor.

To create this structure, the same methods as in socketcan_isotp are implemented, with all the necessary structures available to the user. Resolving into the exact same socket-creating experience as the socketcan_isotp [23] has.

To implement the *read_packet()* and *write_packet()* methods, we need to define Futures these methods return. Those will be *IsoTpWriteFuture* and *IsoTpReadFuture*

```
pub struct IsoTpWriteFuture<'a> {
    socket: &'a IsoTpSocket,
    packet: &'a [u8],
}

pub struct IsoTpReadFuture<'a> {
    socket: &'a IsoTpSocket,
}
```

By implementing the Future trait on these structures, they are ready to be used in the Async/Await context.

The current implementation uses different polling mechanisms for write and for read. The read implementation uses the *try_io* method on the ready_guard, which is the standard way of implementing the poll method.

The *try_io* method can be seen on lines 9-10 of the following code snippet.

```
impl Future for IsoTpReadFuture<'_> {
    type Output = io::Result<Vec<u8>>;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
    -> Poll<Self::Output> {
        loop {
            let mut ready_guard =
```

```
8                    ready!(self.socket.0.poll_read_ready(cx))?;
9                match ready_guard.try_io(
10                           |inner| inner.get_ref().read_to_vec()) {
11                   Ok(result) => return Poll::Ready(result),
12                   Err(_would_block) => continue,
13               }
14           }
15       }
16   }
```

The write method uses a more bare-bone implementation, by calling write and then, if the write is blocking, interacting with the ready_guard manually.

The interaction with the ready_guard is done on line 11 of the following code.

```
1   impl Future for IsoTpWriteFuture<'_> {
2       type Output = io::Result<()>;
3
4       fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
5       -> Poll<Self::Output> {
6           loop {
7               let mut ready_guard =
8                   ready!(self.socket.0.poll_write_ready(cx))?;
9               match self.socket.0.get_ref().write(self.packet) {
10                  Err(err) if err.kind() == io::ErrorKind::WouldBlock => {
11                      ready_guard.clear_ready();
12                      continue;
13                  }
14                  Ok(_) => return Poll::Ready(Ok(())),
15                  Err(err) => return Poll::Ready(Err(err)),
16              }
17          }
18      }
19  }
```

This difference is caused by the error that occurred when calling two consecutive writes, one of which ended with EAGAIN – signaling that the socket is busy and the write would be blocking. The expected behavior is that the execution of the write will resume once the socket is ready to be written into again. However, such behavior did not occur, resulting in a nonresponsive program.

After going through the tokio documentation and looking at the implementation of other async sockets, without finding any solution, I went to the maintainers of the tokio for help [39].

I was directed to look into the behavior of the epoll. And found that the exact same behavior happens when calling the epoll. This resulted in the opening of two bugs for Linux kernel.

First patch [40] created by my supervisor, correcting the behavior of poll, and Second patch [41] repairing the behavour of epoll.

These two patches resolved in the correct behavior not only of my library, but also of the Linux kernel driver.

Since the patch was not yet shipped by the maintainer of the distribution that I am using, I kept the old implementation in place, which works better with the unpatched

Linux kernel. Once the patch is more prominent, the write implementation will also be done using the *try_io* mechanism.

Apart from the file lib.rs, where the async API is located, the library also contains the socketcan_isotp.rs with the slightly changed API.

The Futures, created with the async read or write method, need a reference to the Async IsoTpSocket to execute the I/O operations. If the unchanged socketcan_isotp library were used, the WriteFuture would stay the same, however, the ReadFuture would need to have a mutable reference to the socket.

The reason for that being that the method read in the socketcan_isotp library takes mutable reference to the self as the argument. When we look into the implementation, the reason for that is obvious:

```
pub struct IsoTpSocket {
    fd: c_int,
    recv_buffer: [u8; RECV_BUFFER_SIZE],
}
...
impl IsoTpSocket {
    ...
    pub fn read(&mut self) -> io::Result<&[u8]> {
        let buffer_ptr = &mut self.recv_buffer as *mut _ as *mut c_void;

        let read_rv = unsafe {
            read(self.fd, buffer_ptr, RECV_BUFFER_SIZE)
        };

        if read_rv < 0 {
            return Err(io::Error::last_os_error());
        }

        Ok(&self.recv_buffer[0..read_rv.try_into().unwrap()])
    }
    ...
}
```

The received message is stored in the internal buffer of the IsoToSocket structure. It could be useful when creating a safety-critical application, since it would avoid allocating memory on the heap. However, in our application, the approach with non-mutable reference is preferable, resulting in modifying the function to return the owned variable in the form of *Vec<u8>* instead of just borrowed slice.

4.2 uds-rs

The code discussed in this section is freely available on crates.io [37].

The uds-rs library provides some UDS services to the user. The current project structure can be seen in the Figure 4.1. The library was written primarily to be used with ISO-14229-1 [2] but also supports VW80124 [3]

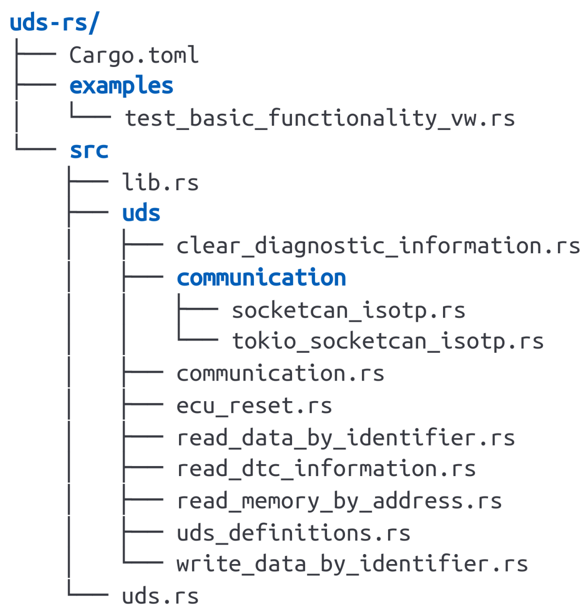

Figure 4.1. Tree view of the uds-rs folder structure

The mandatory lib.rs file is just a wrapper around the uds module, where all the code is located. In the uds.rs file, the *UdsClient* structure is located, which provides the interface for the user. It also holds the private method *send_and_receive* with the communication backend.

As previewed in Background 2.1.1 chapter the basis of UDS is the request-response communication. This is true for the basics, but there are more advanced functionalities, or manufacturer modifications, that expand on this basis. Some services can have subscription-based model, others can suppress positive response, only reporting the negative one. The current implementation adheres strictly to the request response schema, the only deviation being that one transaction can consume multiple negative responses, if they are of type *RequestCorrectlyReceivedResponsePending*.

This approach is easy to implement and is sufficient for the current implementation. However, if the library would support the above-mentioned advanced functionalities, the internal architecture would need to be rewritten.

Each implemented service is placed in its own module with the same design principle. Implement the corresponding service as a method for UdsClient, taking all the necessary data to create the request message as arguments. The return type for all services is *EcuResponseResult* described in the following code snippet. The code is shortened and slightly modified for better readability.

```
1  pub type EcuResponseResult = Result<UdsResponse, UdsError>;
2
3  pub enum UdsResponse {
4      EcuReset(DataFormat<EcuResetResponse>),
5      ReadDataByIdentifier(DataFormat<ReadDataByIdentifierResponse>),
6      ...
7  }
8
9  pub enum DataFormat<T> {
```

```
10        Parsed(T),
11        Raw(Vec<u8>),
12    }
13
14    pub enum UdsError {
15        NRC { nrc: NrcData },
16        UnknownNRC { rejected_sid: u8, unknown_nrc: u8 },
17        UnsupportedSubfunction { unsupported_subfunction: u8 },
18        ...
19    }
```

As we can see, the EcuResponseResult is just a wrapper containing either the UdsResponse or UdsError. UdsError groups all errors that could occur during UDS communication. Please note that NRC is handeled as an error. UdsResponse is an enum containing all the implemented services as its variants, each of which then contains the returned data by the ECU.

As we can see in the snippet, the returned data can be either *Parsed* or *Raw*. Some services, such as *ReadDataByIdentifier* can result in responses that are impossible to parse, with the knowledge of the UDS protocol alone. That is why there is an ability to return the correct, but unparsed, message. If parsed correctly, the corresponding structure containing all the data is placed in variant Parsed.

The service implementation comprises of composing the data from provided arguments sending and receiving the data and parsing the data to the response structure. This approach is the same for all services.

There are two modules, communication and uds_definitions, which do not represent uds services. Uds_definitions, as the name suggests, definitions, and other constants, such as NRC codes or service identifiers. Module communication contains the underlying ISO-TP communication.

One of the design requirements of the uds-rs is that it be usable with other communication protocols, other than ISO-TP. The structure *UdsSocket* in the module communication serves just that purpose, creating a layer between the uds-rs and underlying communication.

I considered using trait *UdsSocket* that would provide async API. The low-level communication struct would be able to implement this trait, and the rest of the uds-rs would use generic *impl UdsSocket* for the interaction.

This approach was scratched, since Rust did not natively support async functions in traits. This changed only recently with the rust 1.75 [42] meaning the future of uds-rs could go in this direction.

4.3 odx-parser

The odx-parser library serves the purpose of deserializing the ODX files from XML to Rust structures. It also provides some essential functions such as resolving odxlinks to enable browsing of the data stored in the ODX file. The library is rather barebone as of the current iteration, not providing any higher-level logic provided by the ODX standard.

Because of the rather unfinished state, it is not yet available as a freely available crate.

4.3.1 Deserialization

There are three main approaches to storing the deserialized data. Either deserialize it into map types, like HashMap, recreate the xml tree using Structures and Enums or combination of those two.

I decided to use Structures and Enums as my approach. The deserialisation into HashMaps would be easier, but implementing additional features would be tedious and error-prone.

It is arguably better for the user of the library, too, since all features of the rust language such as pattern mathing can be used.

To deserialize into structures, the corresponding structures need to be provided. The standard provides the XML format structure in the form of an odx.xsd file. Some reaserch showed that the community support for the XML language in Rust is limited. The most complete library is xgen [43]. However, the code generated by this library is not correct.

For example, the xsd definition:

```xml
<xsd:complexType name="DOP-BASE">
    <xsd:sequence>
        <xsd:group ref="ELEMENT-ID"/>
        <xsd:element maxOccurs="1" minOccurs="0" type="A" name="A"/>
    </xsd:sequence>
    <xsd:attribute use="required" type="xsd:ID" name="ID"/>
    <xsd:attribute use="optional" type="xsd:string" name="OID"/>
</xsd:complexType>
```

Resolves into the following rust code:

```rust
// DOPBASE ...
#[derive(Debug, Deserialize, Serialize, PartialEq)]
pub struct DOPBASE {
   #[serde(rename = "ID")]
   pub id: String,
   #[serde(rename = "OID")]
   pub oid: Option<String>,
   #[serde(rename = "ELEMENT-ID")]
   pub elementid: ELEMENTID,
   #[serde(rename = "ADMIN-DATA")]
   pub a: A,
}
```

Here, the existence of minOccurs=0 maxOccurs=1 is not interpreted as an option and resolves into run-time error when the element *A* will not be encountered. This is just one example of many [1], which have made me scratch this approach. In addition, the deprecated library serde_xml_rs [44] is used for the deserialization.

I used the generated code as a very light inspiration and rewritten odx.xsd to rust manually. If faced with the same decision today, I would probably try to implement a working generator, rather than manually rewriting structures.

The quick-xml [45] library is used for the deserialization. In contrast to the serde_xml_rs, is in active development and is supported.

[1] Other errors are: not working xsd:choice, using #[serde(flatten)] on structs, that cant be flattened, ignoring default values and incorrectly handling xsd:extension

Some workarounds are needed to provide a correct deserialization of the ODX format.

The greatest limiting factor of using Rust to deserialize ODX file is that ODX uses inheritance, which is not implemented for Rust. Serde provides the *flatten* functionality that can be used to complement the inheritance. However, this functionality is partly broken for deserializing the XML code. And only structures containing attributes can be flattened. It is a known issue referenced even in quick-xml documentation [46] which is dependant on the serde issues [47–48].

This bug can be solved by implementing the flattened structure directly into the parent element. Resolving into inelegant, but working code.

Another issue with deserialization is the deserialization of HTML text, since ODX allows HTML code to be used in some places. However, the complex structure of HTML is not really suitable to be deserialized. I wanted to implement this behavior to take the HTML code and put it in the String variable, but due to the bug [49] in quick-xml it is not supported. It could be done by implementing a custom deserializer for the elements containing HTML code, but for now it is just ignored.

With these workarounds, the deserialization of the ODX file works. Few elements are currently not supported, meaning they are ignored when encountered, but most of the ODX elements are correctly deserialized.

4.3.2 Resolving ODX references

With the ODX structure deserialized, a database of all elements containing *ID* is needed to resolve the *ODXLINKs*.

The process is described on *ID*s, but the same was applied for the *SHORT-NAME*s. Currently, the different scopes defined for *SNREF*s are ignored. This trivialization is allowed by the fact that the important elements referenced via *SHORT-NAME* in the provided PDX data are uniqe. However, this needs to be resolved in a later iteration of the library.

The structure of the Database struct looks as follows:

```
pub enum IdSnReference {
    VehicleInfoSpec(Arc<VehicleInfoSpec>),
    InfoComponent(Arc<InfoComponent>),
    ComparamSpec(Arc<ComparamSpec>),
    ...
}
pub struct OdxStructure {
    my_data: Odx,
    import_data: HashMap<String, Odx>,
    parent_refs: HashMap<String, OdxStructure>,
    database: Database,
}

pub struct Database {
    id_database: HashMap<String, IdSnReference>,
    sn_database: HashMap<String, IdSnReference>,
}
```

OdxStructure owns all the data, adding additional references lazily as they are encountered during run-time. This approach is made possible by the usage of smart pointers. The smart pointer used is Arc [50] (Asynchronous reference counter).

Smart pointers have performance overhead over the regular references, but make working with the references and especially lifetimes of the objects much easier.

This also means that at least each element with *ID* in the Odx tree of objects needs to be encapsulated in *Arc<T>*. Currently, only structures containing *ID* or *SHORT-NAME* are encapsulated in Arc.

To create the *Database* the odx-parser library provides *GetDatabase* derive macro. This macro, when used on an object, implements a custom trait *GetDatabase*. The *GetDatabase* trait is described in the following code snippet:

```
trait GetDatabase {
    fn get_database(&self) -> Database;
    fn get_id(&self) -> Option<String>;
    fn get_sn(&self) -> Option<String>;
}
```

The most important is the *get_database* function. The *get_id* and *get_sn* are just helper methods for the *get_database* method.

The macro implements the *GetDatabase* trait in the following matter:

1) Create two HashMaps, *sn_hashmap* and *id_hashmap*. One for elements with ID and one for elements with SN.

2) Go through the fields of structure, or variants of enum. Skip children-less[2] types and call *get_id* and *get_sn* on the rest. If the method returns *Some(x)*, add the x as the key and the reference to the object as the value to the corresponding HashMap.

3) Create Database *d* from the *sn_hashmap* and *id_hashmap*.

4) Go through the fields of structure, or variant of enum once again, calling *get_database* on potential parent[3] types. Merge the returned Database with *d*.

5) Return *d*

The GetDatabase can be implemented for:

1) Structs with named fields. Allowed types are primitives and types that implement the GetDatabase trait. These types can be encapsulated in Vec, Option, or Arc.

2) Enums with variants that have either no data associated or one unnamed tuple struct value. The value can be encapsulated in Arc.

The GetDatabase macro does not currently support namespace paths in the type definitions.

Note that the object we call the *get_database* on does not add itself to the returned database. This is due to the fact that we call *get_database* on *T.get_database()* and not on *Arc<T>.get_database()*. And we need Arc<T> in order to create a reference for the database.

As mentioned in chapter 3.4 the *ODXLINK* refers to other files through the attributes *DOCREF* and *DOCTYPE*. The *DOCTYPE* specifies the top element of the ODX file and the *DOCREF* specifies the *SHORT-NAME* of the element. These two attributes need to be translated into the filename so that it can be deserialized. For this purpose, the OdxPackage structure is created:

[2] children-less type is a primitive. Primitive in this context is not just primitive from the Rust point of view, but also other types that are not ODX types, like Strings

[3] Parent is the oppsite of children-less type.

```
struct OdxFolder {
    layer_ref: HashMap<String, std::path::PathBuf>,
    comparam_ref: HashMap<String, std::path::PathBuf>,
}
```

When created, OdxFolder searches the provided folder path, parses all *.odx files, and creates the entry of *SHORT-NAME* and *std::path::PathBuf* in the corresponding HashMap. The *DOCTYPE* supports five variants, but in the OdxPackage provided, only *LAYER* and *COMPARAM* (referencing DIAG-LAYER-CONTAINER and COMPARAM-SPEC respectively discussed in 2.4) variants are used. That is why only the layer_ref and comparam_ref fields are currently implemented.

Since the creation of the OdxFolder is time consuming, the ability to store and load data to a json folder is implemented.

The library is not yet ready for the public release; even tho the bare bones are working, the final application should also provide API calls that provide a more user-friendly way of interaction. Examples could be as follows; list all the possible diagnostic communications, or automatically parse provided data.

The current implementation provides only essentials that can be used as a building block when implementing more complex features.

Chapter 5
Evaluation

This chapter provides example usage of the libraries presented as well as conditions under which they were tested.

5.1 Testing the functionality of the libraries

Essential testing of the **tokio-socketcan-isotp** library was carried out using an echo server similar to the application in the tokio-isotp-echo [51] repository. Testing using the echo server was performed on both the CAN and Virtual CAN interfaces. The bus monitoring and responses were provide trough can-utils [22] applications, primarily *candump*, *isotpsend* and *isotprecv*.

Most of the real-world tests were carried out on the Bosch 3Q0919283 parking ECU. Communication was also successfully tested on the TSI 1.5 motor control unit.

The functionality of the **uds-rs** library was tested on the same ECUs. In addition to the communication testing, the test modules are added primarily to confirm the correct deserialization from method arguments to packets and serialization from packets to the return types of these methods.

All communication tests were performed on the machine running the Linux kernel 6.6.5.

The **odx-parser** library is tested both on artificial data and the provided real-world ODX database. Artificial data are more directed towards checking the correctness of the individual ODX tree structures and their correct deserialization. The real-world data were used more to test the correct integration of the individual structures.

The artificial data can be found in test modules and in separate files in the odx-files folder. The real-world data are under NDA and therefore they are not published together with this work.

5.2 Creating sample application

To demonstrate the functionality of libraries. I created two simple programs. One for the UDS and the other for the ODX.

5.2.1 Basic CLI UDS application

To show the basic functionality and usage of the uds-rs library, I created a very basic application with command line interface (CLI). The application can be found in the diag-cli folder of the included application. The CLI of the application was created using clap [52] library.

The application provides four UDS services through its interface, *ReadDTC*, *ClearDTC*, *ReadDataByIdentifier*, and *ECUReset*. The user interface supports the

```
Specify which service to use

Usage: diag-cli -s <SOURCE_ADDRESS> -d <DESTINATION_ADDRESS> <CAN_INTERFACE>
<COMMAND>

Commands:
  read-dtc                  ReadDTCInformation - service 0x19
  clear-dtc                 ClearDiagnosticInformation - service 0x14
  read-data-by-identifier   ReadDataByIdentifier - service 0x22
  ecu-reset                 ECUReset - service 0x11
  help                      Print this message or the help of the given
subcommand(s)

Arguments:
  <CAN_INTERFACE>

Options:
  -s <SOURCE_ADDRESS>
  -d <DESTINATION_ADDRESS>
  -h, --help                     Print help
  -V, --version                  Print version
```

Figure 5.1. ./diag-cli --help

```
ReadDataByIdentifier - service 0x22

Usage: diag-cli -s <SOURCE_ADDRESS> -d <DESTINATION_ADDRESS> <CAN_INTERFACE>
read-data-by-identifier --did <DID>

Options:
      --did <DID>
  -h, --help       Print help
```

Figure 5.2. ./diag-cli read-data-by-identifier --help

option `--help` to list all the possible commnads. Result of that listing is in the Figure 5.1.

We can also get more information about specific commands by calling:

`./diag-cli read-data-by-identifier --help`.

The result of that call is in the Figure 5.2

The presented interfrace can be used to, for example, extract the name of the ECU connected using following command:

`./diag-cli -s 0x74 -d 0x7A can0 read-data-by-identifier --did 0xf197`

CAN IDs are fictional, but DID *0xf197* is defined by the UDS standard and can be used to retrieve the name of the ECU. This DID is called *systemNameOrEngineType-*

```
ReadDataByIdentifier(
    Parsed(
        ReadDataByIdentifierResponse {
            data_records: [
                DataRecord {
                    data_identifier: 0xf197,
                    data: [
                        0x50,
                        0x44,
                        0x43,
                        0x20,
                        0x34,
                        0x20,
                        0x4b,
                        0x61,
                        0x6e,
                        0x61,
                        0x6c,
                        0x20,
                        0x20,
                    ],
                },
            ],
        },
    ),
)
```

Figure 5.3. ./diag-cli -s 0x74 -d 0x7A can0 read-data-by-identifier --did 0xf197

DataIdentifier and can be found in Table C.1 in [2]. After executing the program with the Bosch parking ECU connected, we get the message show in the figure 5.3

From the listing, we can see that the message was properly parsed and contains the requested data. If we translated the data received into ASCII characters, we would get: "`PDC 4 Kanal  `".

Since calling of the service is quite verbose, the diag-cli contains Makefile, where all the variables, like can interface, can be predefined.

5.2.2 Interpreting ECU response using an odx-parser

The second example application is aimed more as a proof of concept than a real world application. But it can give the reader an idea of how this library could be built to provide more advanced operations on the ODX files.

The application consists of the `example_parse_input_data.rs` source code file located in odx-parser/examples/ and two ODX files, located in odx-parser/odx-files/example-odx-folder. The ODX files are purely fictional, not representing any real-world vehicle, only providing the bare minimum for us to demonstrate the functionality of the library. The root ODX file is *ecu.odx* where the ECU named *BLOB* is located.

The BLOB provides a single diagnostic service *ReadMesurementData*, which is a simplified modification of the ReadDataByIdentifier service. This service can return data for two DIDs: 0x100 and 0x200. The DID 0x100 provides the temperature in Celsius, and the DID 0x200 provides the voltage in miliVolts.

Units are stored in a separate file *data_library.odx*. This library symbolizes the shared ECU data module of the ODX and serves primarily as a demonstration, how the odx-parser handles references to different files, than root file.

The program assumes that some communication already took place and starts with the *ecu_response* variable, which represents ReadDataByIdentifierResponse, identical to the one previously mentioned CLI application could return.

The following bullet points depict the code behavior:

1. Deserialize the ecu.odx file and go through the files in the folder, record what the ID of the DIAGNOSTIC-LAYER stored in those files is for potential future references.
2. Go through the positive responses of the *BLOB* ECU and find the one that matches 0x62 – the SID of the ReadDatabyIdentifier response.
3. Translate the DID into the *Structure* element, which describes how the data are structured. The translation takes *Table* referenced by the positive response, translates the DID into *Key*, which can then select the corresponding *Table row* where the *Structure is referenced.*
4. Take *Data Object Property* (DOP) referenced by the *Structure* and use *Computation category*, *nominators* and *denominators* to compute the real value of the received Data.
5. Take *Unit* referenced by the DOP, identify to which file the reference points, and deserialize the file. Resolve the reference and take the *Unit representation*.
6. Display the computed value with the unit.

Using the described code, the data on the following code snippet:

```
1  ReadDataByIdentifierResponse {
2      data_records: vec![
3          DataRecord {
4              data_identifier: 0x100,
5              data: vec![0x91],
6          }
7      ],
8  };
```

Can be translated into the value `22.5°C`. If we changed the data_identifier to be 0x200, the second DID supported by the ECU, the result of the process would be `1450mV`.

Despite the fact that the described ODX files are quite trivial, the code of this example is quite long and can be hard to follow. Proving that this would not be sufficient approach of the final application, however, this example can give us a good notion of how the more advanced features of the odx-parser could be implemented.

Chapter 6
Conclusion

As a result of this thesis, three software libraries were created:

- **tokio-socketcan-isotp** library that provides asynchronous ISO-TP communication.
- **uds-rs** library implementing some UDS services also asynchronously.
- **odx-parser** library capable of parsing ODX files and providing some interactions with the ODX database.

The first two are already available to the public as open-source projects[1][2], while the odx-parser library is in the working state with some missing features.

Also provided are simple applications showing the functionality of the implemented libraries and examples of usage.

The goal of this project, the creation of the full graphical diagnostic application, was not achieved, and only the command-line version is available. Even such an application already simplifies some tasks in various development projects. The implemented libraries should greatly simplify the process of creating the final diagnostic application.

6.1 Future work

In the future work, I will improve the odx-parser to be complete enough to be published along the other two libraries as an open-sourse project. The other two libraries, although published, can still be improved upon. The tokio-socketcan-isotp could provide other APIs as discussed in 3.2. There are also additional services that could be implemented for the uds-rs library.

[1] https://crates.io/crates/tokio-socketcan-isotp
[2] https://crates.io/crates/uds-rs

References

[1] Jan Vojnar. *Simulator of TSI 1.5 combustion engine*. Accesssed: Jan. 7, 2024 [Online].
https://dspace.cvut.cz/bitstream/handle/10467/111218/F3-DP-2023-Vojnar-Jan-diploma_thesis_vojnar_jan.pdf?sequence=-1&isAllowed=y.

[2] ISO. *14229-1:2013 Road vehicles — Unified diagnostic services (UDS) — Part 1: Specifiaction and requirements*. International Organization for Standardization. 2013.
https://www.iso.org/standard/55283.html.

[3] Volkswagen Group. *VW 80124 issue 2010-06, Unified Diagnostic Services Protocol, Application Layer & Implementation*. Standards Department of a Volkswagen Group.

[4] ISO. *15765-2:2016 Road vehicles — Diagnostic communication over Controller Area Network (DoCAN) — Part 2: Transport protocol and network layer services*. International Organization for Standardization. 2016.
https://www.iso.org/standard/66574.html.

[5] CAN Specification. Bosch. *Robert Bosch GmbH, Postfach*. Accesssed: Jan. 4, 2024 [Online], 50 15.

[6] *CAN Bus Explained - A Simple Intro [2023*. CAN in Automation. Accesssed: Jan. 4, 2024 [Online].
https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial.

[7] William Bugden, and Ayman Alahmar. *Rust: The Programming Language for Safety and Performance 2022*. Accesssed: Jan. 4, 2024 [Online].
https://arxiv.org/abs/2206.05503.

[8] The Rust Project Developers. *The Rust Programming Language*. Accesssed: Jan. 4, 2024 [Online].
https://doc.rust-lang.org/1.67.1/book/.

[9] *RAII*. Accesssed: Jan. 4, 2024 [Online].
https://en.cppreference.com/w/cpp/language/raii.

[10] Eric Lippert. *Asynchronous Programming - Easier Asynchronous Programming with the New Visual Studio Async CTP*. Accesssed: Jan. 4, 2024 [Online].
https://learn.microsoft.com/en-us/archive/msdn-magazine/2011/october/asynchronous-programming-easier-asynchronous-programming-with-the-new-visual-studio-async-ctp.

[11] The Rust Release Team. *Announcing Rust 1.39.0*. Accesssed: Jan. 4, 2024 [Online].
https://blog.rust-lang.org/2019/11/07/Rust-1.39.0.html.

[12] Niko Matsakis. *Async-await on stable Rust!* Accesssed: Jan. 4, 2024 [Online].
https://blog.rust-lang.org/2019/11/07/Async-await-stable.html.

[13] *Asynchronous Programming in Rust – The Async Ecosystem*. Accesssed: Jan. 4, 2024 |Online|.
https://rust-lang.github.io/async-book/08_ecosystem/00_chapter.html.

[14] Tokio. *Asynchronous runtime for the Rust programming language*. Accesssed: Jan. 4, 2024 |Online|.
https://tokio.rs/.

[15] async-std. *Asynchronous runtime for the Rust programming language*. Accesssed: Jan. 4, 2024 |Online|.
https://async.rs/.

[16] smol-rs. *A small and fast async runtime*. Accesssed: Jan. 4, 2024 |Online|.
https://github.com/smol-rs/smol.

[17] yoshuawuyts. *Asynchronous runtime for the Rust programming language*. Accesssed: Jan. 4, 2024 |Online|.
https://github.com/async-rs/async-std/issues/992#issuecomment-1035223559.

[18] *Documentation on Mio*. Accesssed: Jan. 4, 2024 |Online|.
https://tokio-rs.github.io/mio/doc/mio/.

[19] Dietmar Natterer, Thomas Strobele, and Franz Krauss. *ODX process from the perspective of an automotive supplier 2014*. In: Accesssed: Jan. 4, 2024 |Online|.
https://api.semanticscholar.org/CorpusID:6612764.

[20] *SocketCAN - Controller Area Network*. The Linkx Kernel. Accesssed: Jan. 4, 2024 |Online|.
https://www.kernel.org/doc/html/latest/networking/can.html.

[21] Oliver Hartkopp. *can-isotp*. Accesssed: Jan. 4, 2024 |Online|.
https://github.com/hartkopp/can-isotp.

[22] Oliver Hartkopp, and Marc Kleine-Budde. *SocketCAN userspace utilities and tools*. Accesssed: Jan. 4, 2024 |Online|.
https://github.com/linux-can/can-utils.

[23] Marcel Buesing. *Marcelbuesing/Socketcan-ISOTP: ISO-TP Socketcan Library for rust*. Accesssed: Jan. 4, 2024 |Online|.
https://github.com/marcelbuesing/socketcan-isotp.

[24] Terry Kerr. *Oefd/Tokio-socketcan: Asynchronous linux SOCKETCAN sockets with Tokio*. Accesssed: Jan. 4, 2024 |Online|.
https://github.com/oefd/tokio-socketcan.

[25] Socketcan-Rs. *Socketcan-Rs/SOCKETCAN-Rs: Linux socketcan access in rust*. Accesssed: Jan. 4, 2024 |Online|.
https://github.com/socketcan-rs/socketcan-rs.

[26] Pier-Yves Lessard. *python-udsoncan*. Accesssed: Jan. 4, 2024 |Online|.
https://github.com/pylessard/python-udsoncan.

[27] *Vector Products from A to Z*. Vector Group. Accesssed: Jan. 7, 2024 |Online|.
https://www.vector.com/int/en/products/products-a-z/.

[28] Ashcon Mohseninia. *Rnd-ash/openvehiclediag: A rust based cross-platform ECU diagnostics and car hacking application, utilizing the passthru Protocol*. Accesssed: Jan. 4, 2024 |Online|.
https://github.com/rnd-ash/OpenVehicleDiag.

[29] Ashcon Mohseninia. *year=Accesssed: Jan. 9, 2024 |Online|,Rnd-ash/ecu_diagnostics: A rust crate for ECU diagnostic protocols (UDS / KWP)*. Accesssed: Jan. 4, 2024 |Online|.
https://github.com/rnd-ash/ecu_diagnostics.

[30] *odxtools – set of utilities for working with the ODX standard*. Mercedes-Benz. Accesssed: Jan. 4, 2024 |Online|.
https://github.com/mercedes-benz/odxtools.

[31] *Trait futures::stream::stream*. CAN in Automation. Accesssed: Jan. 9, 2024 |Online|.
https://docs.rs/futures/0.3.30/futures/stream/trait.Stream.html.

[32] *Trait futures::sink::Sink*. CAN in Automation. Accesssed: Jan. 9, 2024 |Online|.
https://docs.rs/futures/latest/futures/sink/trait.Sink.html.

[33] Tokio. *Trait tokio::io::asyncread*. Accesssed: Jan. 9, 2024 |Online|.
https://docs.rs/tokio/latest/tokio/io/trait.AsyncRead.html.

[34] Tokio. *Trait tokio::io::asyncwrite*. Accesssed: Jan. 9, 2024 |Online|.
https://docs.rs/tokio/latest/tokio/io/trait.AsyncWrite.html.

[35] *Tokio - bridging with sync code*. Accesssed: Jan. 4, 2024 |Online|.
https://tokio.rs/tokio/topics/bridging.

[36] *ASAM MCD-2D (ODX) Version 2.0.1*. Association for Standardization of Automation and Measuring Systems. 2005.
https://www.asam.net/standards/detail/mcd-2-d/.

[37] Jakub Jíra. *A asynchronous tokio ISO-TP library build on top of socketcan-isotp*. Accesssed: Jan. 9, 2024 |Online|.
https://crates.io/crates/tokio-socketcan-isotp.

[38] Tokio. *Rust docs for tokio::io::unix::AsyncFd*. Accesssed: Jan. 4, 2024 |Online|.
https://docs.rs/tokio/latest/tokio/io/unix/struct.AsyncFd.html.

[39] *How to corectly poll_write after WouldBlock*. Accesssed: Jan. 4, 2024 |Online|.
https://github.com/tokio-rs/tokio/discussions/5387.

[40] *can: isotp: fix poll() to not report false EPOLLOUT events*. Accesssed: Jan. 4, 2024 |Online|.
https://lore.kernel.org/all/20230331125511.372783-1-michal.sojka@cvut.cz/.

[41] *can: isotp: epoll breaks isotp_sendmsg*. Accesssed: Jan. 4, 2024 |Online|.
https://lore.kernel.org/all/11328958-453f-447f-9af8-3b5824dfb041@munic.io/.

[42] Tyler Mandry onbehalfof The Async WorkingGroup. *Announcing ‘async fn‘ and return-position ‘impl Trait‘ in traits*. Accesssed: Jan. 5, 2024 |Online|.
https://blog.rust-lang.org/2023/12/21/async-fn-rpit-in-traits.html.

[43] Ri Xu. *XSD (XML Schema Definition) parser and Go/C/Java/Rust/TypeScript code generator*. Accesssed: Jan. 4, 2024 |Online|.
https://github.com/xuri/xgen.

[44] Ingvar Stepanyan. *xml-rs based deserializer for Serde*. Accesssed: Jan. 4, 2024 |Online|.
https://docs.rs/serde-xml-rs/latest/serde_xml_rs/.

[45] Johann Tuffe. *Rust high performance xml reader and writer*. Accesssed: Jan. 4, 2024 |Online|.
https://docs.rs/quick-xml/latest/quick_xml/.

[46] *Module quick_xml::de – Sequences (xs:all and xs:sequence XML Schema types)*. Accesssed: Jan. 9, 2024 |Online|.
`https://docs.rs/quick-xml/latest/quick_xml/de/#sequences-xsall-and-xssequence-xml-schema-types`.

[47] *Serde issue #1905 – Allow to flatten sequences/tuples*. Accesssed: Jan. 4, 2024 |Online|.
`https://github.com/serde-rs/serde/issues/1905`.

[48] *quick-xml issue #326 – Using flatten and rename="$value" on adjacent fields causes error*. Accesssed: Jan. 4, 2024 |Online|.
`https://github.com/tafia/quick-xml/issues/326`.

[49] *quick-xml issue #257 – Help deserialize mixed tags and string in body $value (html text formatting)*. Accesssed: Jan. 4, 2024 |Online|.
`https://github.com/tafia/quick-xml/issues/257`.

[50] *Rust – Struct std::sync::Arc*. Accesssed: Jan. 4, 2024 |Online|.
`https://doc.rust-lang.org/std/sync/struct.Arc.html`.

[51] Jakub Jíra. *Basic echo server built with tokio-socketcan-isotp*. Accesssed: Jan. 4, 2024 |Online|.
`https://github.com/japawBlob/tokio-isotp-echo`.

[52] *Command Line Argument Parser for Rust*. clap-rs. Accesssed: Jan. 7, 2024 |Online|.
`https://docs.rs/clap/latest/clap/`.

Appendix A
Abbreviations

API	Application Programming Interface
ASAM	Association for Standarsidation of Automation and Measuring Systems
ASCII	American Standard Code for Information Interchange
CAN	Controller Area Network
CLI	Command line interface
DID	Data Identifier
DTC	Diagnostic Trouble Code
ECU	Electronic Control Unit
ISO	International Organization for Standardization
ISO-TP	Transport Layer ISO 15765-2
LIN	Local Interconnect Network
LSB	Least significant byte
MSB	Most significant byte
NDA	Non-disclosure agreement
NRC	Negative Response Code
ODX	Open Diagnostic data eXchange
PDX	Packaged ODX
RAII	Resource acquisition is initialization
SID	Service Identifier
UDS	Unified Diagnostic Services
UML	Unified modeling language
W3C	World Wide Web Consortium
XML	eXtensible Markup Language