

University of Reading
Department of Computer Science

Open source vehicle ECU diagnostics and testing platform

Ashcon Mohseninia

Supervisor: Julian Kunkel

A report submitted in partial fulfilment of the requirements of
the University of Reading for the degree of
Bachelor of Science in *Computer Science*

April 29, 2021

Declaration

I, Ashcon Mohseninia, of the Department of Computer Science, University of Reading, confirm that all the sentences, figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly.

Ashcon Mohseninia
April 29, 2021

Abstract

With the complexity of electronics in consumer vehicles, there currently only exists proprietary tools produced by various OEMs to diagnose their own vehicles. Each OEM has its own tool, and there is no easy way for a consumer to diagnose their own vehicle.

This project will explore the possibility of creating an entirely open source diagnostics software stack that will work with all ready existing diagnostic adapters that utilize the Passthru API (Which is used for a PC to communicate with a diagnostic adapter plugged into a vehicles OBD-II port). Additionally, this project will also explore creating an entirely open source Passthru API driver for an open source OBD-II adapter, whilst additionally porting the API from Win32 only to UNIX based operating systems such as Linux and OSX, allowing for a wider target audience compared to traditional diagnostic applications and adapters which only target Windows.

Acknowledgements

For their contributions to this project, I would like to acknowledge the following parties:

- Julian Kunkel - For his supervision of the project and consistently giving me tips on how I can improve various aspects of this report.
- Pat Parslow - For his supervision of the project
- Macchina CC <https://www.macchina.cc/> - For providing great support for their M2 hardware, and for shipping me several test modules as well as an OBD Breakout board for testing with the project.
- JinGen Lim <https://github.com/jglim> - For his efforts with reverse engineering the CBF file format
- Collin Kidder <https://github.com/collin80> - For consistent support with his due_can library used for CANBUS communication on the M2 hardware.
- Daniel Cuthbert <https://github.com/danielcuthbert> - For helping me test both the M2's driver and application on Mac OSX as well as designing OpenVehicleDiag's logos.
- Nils Weiss <https://github.com/polybassa> - For providing me with a preview of Nils Weiss, Sebastian Renner, Jürgen Mottok, Václav Matoušek (n.d.) prior to its official publication.
- RJ Automotive <https://www.rjautomotive.net/> - For allowing me to test my Passthru adapter with their copy of DAS/Xentry

Contents

1	Introduction	1
1.1	Background	1
1.2	Problem statement	2
1.2.1	Lack of continuity or standards between OEMs diagnostic tools	2
1.2.2	Proprietary diagnostic hardware	2
1.3	Aims and objectives	2
1.4	Solution approach	3
1.4.1	JSON schema designing and converting	3
1.4.2	Passthru driver creation	4
1.4.3	Application creation	4
1.5	Summary of contributions and achievements	4
1.5.1	Passthru driver	4
1.5.2	JSON Schema	5
1.5.3	Diagnostic Application (OpenVehicleDiag)	5
1.6	Organization of the report	5
2	Literature Review	6
2.1	Communication protocols found in vehicles	6
2.1.1	CAN	6
2.1.2	ISO-TP	8
2.1.3	LIN	9
2.2	Diagnostic adapter hardware and APIs	12
2.2.1	Hardware APIs	12
2.2.2	Hardware adapters	13
2.3	ECU Diagnostic protocols	14
2.3.1	OBD-II	15
2.3.2	KWP2000 and UDS	17
2.4	Existing diagnostic software	18
2.4.1	Torque for Android (Generic OBD)	18
2.4.2	Carly (Third party software)	18
2.4.3	Xentry (Dealer software)	18
2.5	Open Diagnostics eXchange (ODX)	19
2.6	The OBD-II port	19
2.7	Comparisons to the proposed project	20
2.7.1	Hardware adapters	20
2.7.2	Diagnostic software	21

3 Methodology	22
3.1 Test setup	22
3.2 Rust	23
3.3 JSON schema creation and CBF parsing	23
3.3.1 JSON structure	23
3.3.2 Code implementation of the JSON Schema	27
3.3.3 Parsing Daimler CBF Files to JSON	29
3.4 Cross platform Passthru adapter	32
3.4.1 Architecture	33
3.4.2 Creating the driver in Rust	34
3.4.3 Communication between the adapter and driver	35
3.4.4 Reading battery voltage	37
3.4.5 ISO-TP Communication	38
3.4.6 Porting the Passthru API to Linux and OSX	39
3.4.7 Logging activity	41
3.4.8 Performance optimizations with CAN Interrupts	41
3.5 Diagnostic GUI	42
3.5.1 Diagnostic server architecture	43
3.5.2 Communication server architecture	43
3.5.3 Implementation of the Passthru API	45
3.5.4 User Interface	47
3.5.5 KWP2000 and UDS implementations	53
3.5.6 Automated ECU Scanner	55
3.5.7 JSON Diagnostic session	59
3.6 Summary	62
4 Results, Discussion and Analysis	63
4.1 Passthru driver	63
4.2 Diagnostic application	64
4.2.1 Automated ISO-TP Scanner	64
4.2.2 Diagnostic session mode (JSON)	66
4.2.3 Generic diagnostic session	67
4.3 Summary	68
5 Conclusions and Future Work	69
5.1 Conclusions	69
5.2 Future work	69
6 Reflection	71
Appendices	74
A Screenshots and diagrams	74
A.0.1 Daimler Xentry software	74
A.0.2 OpenVehicleDiag	77
B Issues and resolutions	84
B.1 A full list of issues encountered during development of the M2 driver	84
B.1.1 Windows Serial API	84
B.1.2 CAN Due library	84

C Code snippets and tables	85
C.1 List of SAE J2534 API functions	85
C.2 A full list of driver message types	85
C.3 List of KWP2000 and UDS Services	87
C.4 Extract of OVD's JSON (EGS52) from CBFParser	89
D A list of project repositories	93
E OpenVehicleDiag JSON Schema	94

List of Figures

2.1	Bit layout of both Standard and Extended CAN Frames	6
2.2	Data format of a service request with PID and data	14
2.3	Data format of a positive ECU response	14
2.4	Data format of a negative ECU response	14
3.1	Mock-up of how the ECUs are connected in the test setup	22
3.2	UML representation of ovdECU Root object, ECUVariantDefinition and Connection properties	24
3.3	UML representation of JSON Service and ECUDTC	24
3.4	UML representation of JSON DataFormat	25
3.5	Simplified UML representation of the data structure in a CBF File	30
3.6	Macchina's M2 Under the dash OBD-II module	32
3.7	Macchina M2 board layouts	33
3.8	Expanded sequence diagram of communication server	36
3.9	Voltage reading comparison between M2 (Stock and corrected) and Multimeter	37
3.10	Sequence diagram for sending ISO-TP Data to an ECU	38
3.11	Sequence diagram for receiving ISO-TP Data from an ECU	39
3.12	Diag servers UML overview	43
3.13	UML of ComServer	44
3.14	OpenVehicleDiag's launcher (Passthru device enumeration)	49
3.15	OpenVehicleDiag's launcher displaying Passthru error	50
3.16	Standard CAN display (Hex) vs Binary CAN	53
3.17	Warning message presented to the user prior to the ECU scan	55
3.18	Listing to existing CAN traffic	56
3.19	Locating potential ISO-TP endpoints	56
3.20	Finalizing ISO-TP scan results	57
3.21	Scan progress for UDS compatible ECUs	58
3.22	Instrument cluster warning lights being displayed during the final stages of ECU detection	58
3.23	Results page of ECU Scanner	59
3.24	JSON Session home with CRD ECU	60
4.1	DAS utilizing the custom J2534 adapter	63
4.2	Vediamo utilizing the custom J2534 adapter to flash a ECU	64
A.1	Xentry Diagnostics establishing communication with all the ECUs within a vehicle. During this stage of diagnostics, Xentry is trying to locate all the ECUs on the vehicle, and checking what variation each ECU is in order to parse their diagnostic data correctly	74

A.2	Xentry diagnostics with a list of all possible ECUs in the vehicle to talk to, each in their own category	75
A.3	Obtaining advanced data from the ECU in Xentry - Querying various attributes about the ESP ECU. The serial number of this part has been hidden.	75
A.4	Xentry showing a live 'actuation' value of certain items the ESP ECU controls	75
A.5	Show Xentry obtaining real-time data from the CDI Engine ECU. The values in Green are within tolerance, and values in red are outside tolerance. Black values indicate no tolerances are specified for the value	76
A.6	Xentry showing advanced real-time data from the CDI Engine ECU. This allows for advanced analytics of how the engine is performing.	76
A.7	More advanced real-time diagnostics with the CDI Engine ECU. This shows the injector calibration values for the number 1 cylinder	77
A.8	OVD Home page (Dark theme)	77
A.9	OVD Home page (Light theme)	78
A.10	OVD Can Scanner (Hex mode)	78
A.11	OVD Can Scanner (Binary mode)	78
A.12	Loading a ECU Scan save file in OVD	79
A.13	Selected CRD ECU in OVD	79
A.14	OVD KWP2000 generic session - Home	79
A.15	OVD KWP2000 generic session - Scanning DTCs	79
A.16	OVD KWP2000 generic session - Clearing DTCs	80
A.17	OVD KWP2000 generic session - Sending valid manual payload	80
A.18	OVD KWP2000 generic session - Sending invalid manual payload	80
A.19	OVD Json session - Connected to CRD engine ECU	80
A.20	OVD Json session - ECU Info page	81
A.21	OVD Json session - DTC page	81
A.22	OVD Json session - DTC page with Freeze frame interpretation	82
A.23	OVD Json session - Selecting function to read data from	82
A.24	OVD Json session - Data presentation	83
A.25	OVD Json session - Reading data from EGS52 transmission ECU	83

List of Tables

2.1	Transmission with collision detection on CANBUS	8
2.2	Encoding example of supported PIDs for Service 01	15
2.3	Comparison between existing adapters and proposed solution	20
2.4	Comparison between existing diagnostic software and proposed solution	21
3.1	ECU List in the desk test setup	22
3.2	Supported format options for DataFormat	28
4.1	Automated scan results on the Mercedes W203	65
4.2	Automated scan results on the Mercedes W246	65

List of Abbreviations

CAN/CANBUS	Controlled area network
LIN/LINBUS	Local interconnect network
SAE J2534	The Passthru API
KWP2000	Keyword protocol 2000
UDS	Unified diagnostic services
API	Application programming Interface
ECU	Engine/Electronic control unit
OBD	On-board diagnostics
OBD-II	OBD protocol
ODX	Open Diagnostic eXchange format
DTC	Diagnostic Trouble Code
SCN	Software Calibration Number

Chapter 1

Introduction

In this chapter the current state of car diagnostics will be discussed, as well as a high level overview of the project breakdown and how it will hopefully change the current state of car diagnostics for consumers.

1.1 Background

Since the early 2000s, the automotive industry has seen an exponential increase in both the complexity of ECUs and the number of ECUs in consumer vehicles, with modern vehicles having more than 30 ECUs.

With increased complexity comes more points of failure. With modern ECUs being able to register fault codes at the slightest hint of trouble, and also requiring specialist software to calibrate them after certain mechanical parts are either replaced or modified on a vehicle.

This presents a unique problem. While DIY consumers have traditionally been able to easily replace or modify components on their vehicles, software issues such as ECU fault codes still require proprietary software which is only used by the OEM itself, or licensed to specific workshops as a huge premium, and also requires proprietary multiplexer hardware to plug into the cars OBD-II port, which is also expensive.

Currently, there are simple OBD-II applications that can only communicate with the engine ECU in a vehicle to clear standard OBD-II error codes or read sensor data that is only outlined by the OBD-II specification, but there is no easily available software outside of the OEM's own software (Which is licensed to workshops) which can diagnose all ECU's within a vehicle, or run more complex diagnostic routines and tests. Also, the vast majority of OEM software currently available is only compiled for win32 (32bit Windows), and therefore is not up to date with modern computing, and also does not run on Linux or OSX. This is primarily because OEMs over the years have simply been adding features to their old diagnostic software, rather than spending resources on creating a whole new platform for more modern vehicles.

Therefore, in this project, the possibility and process of creating such an application that allows for communicating with all ECUs within a vehicle, and to run more advanced diagnostic functions on them, without the OEM's own software will be explored. This will also include writing a open source driver for Macchina's M2 OBD-II module to turn it into a diagnostic adapter using the J2534 / Passthru API, which can be used by any application utilizing the protocol, including some OEM software (Such as Daimler's Xentry Passthru diagnostic suite). Additionally, the application and J2534 API will be ported to both 64bit versions of Linux and OSX unofficially, allowing for a much wider target audience of the system, since individuals would no longer be limited to just older versions of Windows.

Another objective of this project is to create a simple, easy to use database format in

JSON. Traditionally, OEM software uses proprietary binary based file formats to describe how the software communicates with ECUs within a vehicle, as well as how to interpret the ECUs responses.

1.2 Problem statement

This section will discuss the current issues with car diagnostics.

1.2.1 Lack of continuity or standards between OEMs diagnostic tools

With every OEM creating their own diagnostic software, there is no continuity between OEM's, which hinders most independent workshops and consumers who wish to diagnose their own vehicles. For instance, Mercedes' diagnostic software (Xentry) will never work on a Toyota vehicle, and vice versa, despite the fact that at a low level, both software suits will use the same hardware API to talk to a diagnostic adapter plugged into a vehicle.

To add to this, there is no standard format for storing diagnostic data about ECU's. Each diagnostic tool set has its own file format, which has to be reverse engineered to extract any useful data from. In this project, the prospect of using an open, universal JSON format will be looked into.

1.2.2 Proprietary diagnostic hardware

Currently, there are two publicly available API's for communicating with a vehicle using a multiplexer. ISO 22900-2 (D-PDU API) and SAE J2534 (Passthru API). Most OEM software supports either one or both of these APIs, or also supports their own proprietary hardware. There are currently two main issues with both of the APIs.

1. Windows only support. Since these API's are designed for diagnostic software, and those are only designed for windows, there is currently no known diagnostic API that includes support for Linux or OSX.
2. Closed source. Although some of the API documentation is made public, vendors of the diagnostic multiplexers that utilize either API creates proprietary hardware with closed-source controller firmware, and sells the adapters at a premium, making it hard for the average consumer to easily attain one. There are chinese 'clone' adapters that can be purchased on ebay for a cheap price, however these tend to not work or will encounter massive stability issues, so should never be trusted to work reliably.

1.3 Aims and objectives

Aims: This project has the following aims:

- Build a cross-platform, Graphical ECU diagnostic application (Using the J2534 passthru API)
- Port the J2534 API to Linux and OSX
- Write a custom J2534 driver for Macchina's¹ M2 Under the dash OBD-II module, allowing it to work on all 3 operating systems.

¹macchina.cc

- Define a JSON schema for describing the capabilities, fault codes, and diagnostic functions that can be ran on an ECU, and make it a viable replacement to proprietary data formats.

NOTE: ECU firmware Flashing or updating will not be part of this project. This is due to liability concerns of leaving an individuals ECU in a bricked state, and also the difficulty in locating a legitimate software version for an ECU.

Objectives: To achieve the aims, the project has the following objectives:

- Read and clear non standard (OBD-II) ECU error codes from a test ECU, and a real vehicle
- Convert Daimler database files (CBF) into JSON as a proof of concept.
- Show that live data recording works on multiple ECU's in a vehicle
- Show that the custom J2534 compatible adapter works with real OEM software that uses the J2534 API

1.4 Solution approach

Breaking down the full solution of the project yields the following sub objectives:

1.4.1 JSON schema designing and converting

For this, a JSON format will be described using UML, then written in code which can be serialized and deserialized to and from JSON. After, a parser will be written which is capable of converting Daimler's CBF file format to the JSON. CBF is a proprietary diagnostic container file format used by Daimler's diagnostic software suits, and is only used for older vehicles (Pre 2008), with their SMR-D file format superseding CBF, which is used by their newer software called Xentry. At a high level, CBF files contain the following data:

1. ECU software version names (An ECU can have different software versions)
2. Communication protocols to be used to communicate with the ECU
3. Payloads to send to the ECU for certain functions
4. List of all error codes, as well as a readable description of the error codes
5. Interpretation data for converting an ECU response packet into human readable text

Important. Since this will extract data from Daimler's own CBF files, there are some social and legal issues to account for. Contained within the CBF files is data related to SCN coding. SCN Coding (Software Calibration number coding), is a way to write a coding string to an ECU in order to enable or disable features on it. The CBF files contain data regarding which regions in the ECU's EEPROM relate to which features. Because SCN coding is something that Daimler sees as its own intellectual property, and charges a heavy fee for feature unlocking on their cars, this will NOT be something that is going to be extracted from the CBF files, and there will be no referencing to SCN regions in the extractor codebase either. The only things that will be extracted from the CBF files will be diagnostic routines, ECU identification data, and interpretation data.

1.4.2 Passthru driver creation

This component of the project will be attempted in the following steps.

1. Define a new standard for storing J2534 configuration data on Linux and OSX, since the J2534 API officially only supports win32 (32bit Windows).
2. Create a reliable cross-platform serial communication link between a PC and adapter, along with a defined protocol for the adapter and PC to exchange data.
3. Create a Rust library with the necessary exported J2534 functions such that any application can use the library and therefore adapter to talk to a vehicle
4. Create C++ firmware for the adapter for managing physical communication links between the OBD-II port and ECUs in the vehicle, and listen for command requests from the PC and sending data back to the PC.

1.4.3 Application creation

This will be the largest part of the project. At a high level, this application has to be able to do the following, and shall be called OpenVehicleDiag:

1. Create an application that can utilize the J2534 API to communicate with a vehicle.
2. Create an abstraction layer for future use, which allows for more hardware to be utilized by OpenVehicleDiag other than J2534. (Examples: SocketCAN, D-PDU).
3. Create a useful GUI for data logging of ECUs based on the data found in the JSON schema
4. Allow for basic commands to be sent to any ECU in a vehicle using KWP2000 or UDS. This should allow for reading and clearing error codes from the vast majority of vehicle ECUs, even if the vehicle does not have any JSON created for it.
5. Allow for a user to see raw data on their vehicles CAN Network (Targeted at individuals who wish to reverse-engineer their vehicles' CAN network)
6. Based on the work done by [Nils Weiss, Sebastian Renner, Jürgen Mottok, Václav Matoušek (n.d.)], create a intuitive user interface which can exploit the ISO-TP protocol to scan for all UDS or KWP2000 ECU's in any unknown vehicle.

1.5 Summary of contributions and achievements

All the code repositories for each part of this project can be located on github (See D). This project has ended up being very successful, gaining a large following online (150 stars on Github for the main OpenVehicleDiag repository). OpenVehicleDiag itself will continue to receive contributions and improvements in the future.

1.5.1 Passthru driver

The Passthru driver implementation has proven that adapters from the likes of Bosch are far too expensive, and there is no need for \$1000+ adapters, when an open source alternative which was designed for this project does exactly the same job with open source hardware

which costs a fraction of the commercial adapters. This report will even show that the adapter works with Commercial software from Daimler. Also, this report will show that the SAE J2534 API can indeed be ported to other operating systems, which in turn makes it easier for other operating system users to utilize the API with custom Diagnostic software such as OpenVehicleDiag.

1.5.2 JSON Schema

The JSON Schema has shown that it can be a compatible substitute for the ODX-D data storage format and other proprietary diagnostic container formats, and more importantly, is a lot easier and more accessible for users to read or create.

1.5.3 Diagnostic Application (OpenVehicleDiag)

OpenVehicleDiag has proven that the need for large scale OEM or expensive third party software is somewhat obsolete, when it comes to simple diagnostics such as DTC reading and clearing, as well as data gathering from the ECU. This should be enough for 90% of use cases where someone would take their vehicle to the dealer due to a software fault with an ECU. It has also proven that diagnostic applications can be intuitive for individuals to use, and can also run on other operating systems other than Windows.

1.6 Organization of the report

This report will first describe the current hardware, software and protocols currently utilized for car diagnostics, before explaining the solution approach and methodology. Each sub project (Passthru driver, JSON Schema, Diagnostic application), will have its own solution approach and methodology.

Towards the end of this report, the results and impact of the work will be discussed, as well as showing tests conducted with the custom Passthru adapter to prove it is SAE J2534 compliant, and works perfectly with commercial diagnostic software. Also future plans for this project will be discussed.

Chapter 2

Literature Review

In this chapter, existing vehicle communication protocols, diagnostic protocols, hardware APIs and diagnostic software will be looked at, with a comparison at the end comparing current diagnostic software and hardware to what is proposed for this project.

2.1 Communication protocols found in vehicles

Within modern vehicles, there are many different protocols used for allowing ECUs in a vehicle to communicate with each other, or to allow an ECU to communicate with primitive components on the car. In this part, the 2 most common protocols (CAN and LIN) will be discussed, outlining how each protocol works, and what they are used for.

2.1.1 CAN

CAN / CANBUS is a high speed transport network used for ECU communication. It consists of 2 separate ISO specifications:

1. ISO11898-2 - High-Speed CAN (Up to 1Mb/s)
2. ISO11898-3 - Low-Speed CAN (Up to 125Kb/s), also known as fault-tolerant CAN

Both CAN Specifications work on similar principles, except with different electrical properties.

CAN Networks transmit CAN Packets. These are data packets containing up to 8 bytes of data, as well as a 11 or 29bit Identifier ID (Depending on if the CAN Network uses Standard or Extended addressing). There is also extra data in each CAN Frame (Bit stuffing and CRC Checks), however these extra bits are never exposed to the ECUs CPU as the CAN Controller deals with validating the CRC checks of the CAN Frame.

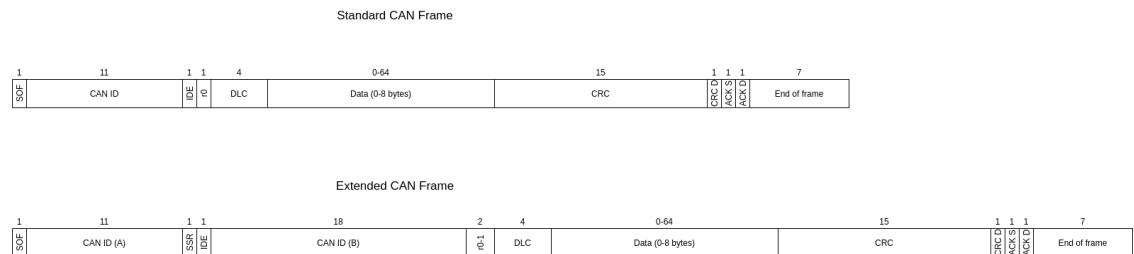


Figure 2.1: Bit layout of both Standard and Extended CAN Frames

Most commonly, the following data fields are exposed the ECU's CPU:

- CAN ID - This is the Unique identifier of the ECU subsystem which transmitted the frame (Some ECUs transmit multiple CAN Frames with different IDs)
- CAN DLC - The amount of bytes in the data portion of the CAN Frame
- CAN Data - 0-8 bytes of data contained within the CAN Frame
- RTR - Remote frame. This is sometimes used by an ECU to request data from another ECU. If this value is 1, then the ECU is requesting data from another ECU, if the value is 0, then there is data within the frame from the requested ECU.

Electrical properties of CAN

Depending on the type of CAN Network, the electrical properties of the bus differs slightly:

- High-Speed CAN ISO11898-2 - Both CAN wires are terminated with a 120Ω resistor at each node on the bus. The recessive voltage of the CAN Network is $2.5V$, with the dominant voltage being approximately $3.5V$ for CAN-H, whilst being approximately $1.5V$ for CAN-L.
- Low-Speed CAN ISO11898-3 - CAN wires are not terminated with a resistor, however the overall resistance between both wires over the entire bus should not exceed 100Ω . The recessive voltage of the CAN Network is approximately $0V$ for CAN-H and $5V$ for CAN-L, whilst the dominant voltages are approximately $5V$ for CAN-H and $0V$ for CAN-L.

Both CAN Network types require that a logical '1' is the recessive voltage, and a logical '0' is the dominant voltage. The logical state of the CAN Network is calculated by applying a logical AND to both the CAN-H and CAN-L wires. Components on the CAN Networks (CAN Transceiver chips) are designed to handle anywhere between $-27V$ to $+40V$ without sustaining any damage. If both CAN Wires are not in the same logical state at the same time EG: CAN-H being in a dominant state whilst CAN-L is in a recessive state, all transmission on the network will stop immediately as all the CAN Transceiver chips on the network as detected that either a wire may be shorted to ground, or that a wire may be shorted to an ECU's power supply, and therefore the network is in an unstable state.

This is not always the case however, as some CAN Networks based on ISO11898-3 can actually run in single wire mode, where the transceivers discard the erroneous wire voltage, and only use the voltage provided by the 'good' CAN wire.

Collision detection in a CAN Network

Since a CAN Network does not have 'master ECU', CAN networks have a passive method of detecting and avoiding packet collisions. As a logical '0' is a dominant voltage, it means that whilst an ECU is transmitting a '0', it is physically impossible for another ECU to flip the CAN Wire voltages to their recessive '1' state. During a frame transmission, each CAN Transceiver chip on the network checks the CAN Wire voltages after every bit is sent, and will stop sending if the CAN Wire voltages do not match the bit being sent in order to avoid a collision. This is why on a CAN Network, ID's with a lower value have a higher 'priority' on the network, since lower ID's will block the transmission of higher IDs in the event of a collision.

CAN ID Bits	10	9	8	7	6	5	4	3	2	1	0	Rest of frame
ECU 1 (CAN ID 0x0005)	0	0	0	0	0	0	0	0	1	0	1	...
ECU 2 (CAN ID 0x000A)	0	0	0	0	0	0	0	1	STOP TRANSMITTING			
CANBUS logical state	0	0	0	0	0	0	0	0	1	0	1	...

Table 2.1: Transmission with collision detection on CANBUS

Table 2.1 shows how an ECU with a lower CAN ID (0x0005) can stop another ECU with a larger CAN ID of (0x000A) from transmitting. As soon as ECU transmitting the higher CAN ID detects the CAN networks logical state does not match that of the bit it just sent, it stops sending, and the data ECU 1 sent is not impacted. This is all handles by the CAN Transceiver chip, so the ECU's Processor is not occupied with checking the CAN network state.

2.1.2 ISO-TP

ISO-TP (ISO15765-2) is a transport layer protocol that runs over a CAN Network, allowing for multiple ECUs to exchange up to 4096 bytes using multiple structured CAN Frames.

The ISO-TP Standard defines the following 4 frame types, which are denoted by the first nibble (Half byte) in the CAN Frame, also known as the PCI byte:

CAN frame type	Nibble (Hex)	Description
Single frame	0	The single frame contains all the data in the payload
First frame	1	This is sent if the data is larger than a single CAN Frame can contain. It also contains the full length of the payload to be transmitted
Consecutive frame	2	Contains subsequent data for a larger data payload after a first frame was sent and the sender received a flow control message.
Flow control	3	This is sent in response to receiving a First frame. This frame tells the sending ECU how many CAN frames to send next before waiting for another flow control message, as well as how quickly to send the frames

A flow control frame contains the following 3 bytes:

Byte name	Byte position	Purpose
PCI	0	Identifies the frame as being a flow control frame. A PCI of 0x30 indicates that it is a flow control frame, and that the sender can proceed sending data, and respect the values indicated in BS and ST-MIN. A value of 0x31 indicates the ECU is currently busy, and the sender must wait before attempting to send again.
BS	1	The BS value (Block Size) indicates the maximum number of consecutive frames the sender can send before the receiving ECU sends another flow control frame to the sender. A value of 0x00 for block size indicates an infinite block size, so the sender must send all its data without waiting for another flow control message.
ST-MIN	2	This indicates the minimum separation time between the sender sending consecutive frames to the receiving ECU. Any value between 0x01 and 0xF0 is interpreted as milliseconds, and values between 0xF1 and 0xF9 is interpreted as 100-900 microseconds. A value of 0x00 here indicates the sender can send consecutive frames as quickly as it wants.

In order to identify lost data, or out of order CAN frames during multi frame transmissions, the PCI byte of the consecutive frame is used as a counter. The counter starts with a PCI of 0x21, then increases after every consecutive frame is sent until 0x2F, before wrapping round to 0x20 and counting up again.

Below is an ISO-TP exchange traced from the interior CAN Network of my Mercedes W203 C class. This exchange occurred between the Radio and Instrument cluster, where the Radio is telling the instrument cluster what text to display on the Radio page of its LCD. Red indicated bytes that are for the ISO-TP protocol

CAN ID	CAN Data								ASCII
0x01A4	10	12	03	26	01	00	01	0B
0x01D0	30	08	28	00	00	00	00	00	0.(....
0x01A4	21	10	41	55	44	49	4F	20	!.AUDIO.
0x01A4	22	4F	46	46	00	C4	00	0B	"OFF....

In this trace, we can see the Radio has a CAN ID of 0x01A4, and sent a 18 byte payload to the instrument cluster. Upon receiving the first CAN Frame, the instrument cluster responded on a CAN ID of 0x01D0, accepting the transmission of more data, and asking for a maximum of 8 packets to be sent before the radio should wait for another flow control message, and with each packet being sent at a minimum of 40ms apart. Reassembling the Radio's payload without any of the ISO-TP protocol bytes shows the raw payload:

```
03 26 01 00 01 0B 10 41 55 44 49 4F 20 4F 46 46 00 C4
```

2.1.3 LIN

LIN (Local interconnect Network) is a much cheaper and simpler interconnect network used in vehicles. Its main focus is for allowing ECUs to drive more primitive components simply, rather than using expensive network solutions such as CANBUS.

A LIN network consists of a single wire, which connects 1 master ECU with up to 16 slave components or ECUs. This single wire acts as a half-duplex serial interface, running at 12V DC. The data transfer rate of LIN Bus is relatively slow, ranging from 1-20kbps. However, due to its simplicity, LIN bus is preferred when driving primitive components. For example, the engine ECU in a vehicle can use a LIN network to drive items like the engine fan and AC

compressor on the engine, allowing the engines ECU to also get feedback on the components state of operation.

A LIN network operates by the master ECU requesting data from a slave component on its network. The network is otherwise void of any traffic. When requesting data, the master ECU will send a header frame which looks like the following:

Field name	Length (Bits)	Purpose
Sync break	14+	Indicates the start of a new frame
Sync byte	8	Allows for resynchronization of slave nodes
ID byte	6	Frame Identifier
Parity	2	parity for the frames ID

As seen above, the ECU starts by sending a sync break. This notifies all the nodes on the LIN network to begin listening for incoming data and stop transmitting. Following this is the sync byte, This has a predefined value of 0x55 (01010101 in binary) and allows all the nodes on the network to calculate and determine the time between the high and low voltages on the bus, allowing them to listen in sync with the master sending the rest of the frame. Lastly is the ID followed by 2 bits for parity. This is the ID the ECU is requesting data from, and the matching node on the network will respond with a data frame, which is structured like so:

Field name	Length (Bits)	Purpose
Data	1-64	Data to be transmitted
Checksum	8	Checksum calculation

The response sent back to the master ECU contains simply a data field (configurable between 1 and 64 bits) followed by an 8 byte checksum. Since LIN 2.0, the ID of the request determines the data size of the response:

ID Range	Data length
0-31	2 bytes
32-47	4 bytes
48-63	8 bytes

When calculating the checksum of the frame, the following code is used:

```

1  uint8_t calculate_checksum(uint8_t id, uint8_t* data_ptr, uint8_t data_len,
2    uint8_t cs_mode) {
3      uint16_t tmp = 0; // 0 is default for classic checksum
4      if (cs_mode == ENHANCED) { tmp = id; } // Enhanced checksum (LIN 2.0)
5      for (int i = 0; i < data_len; i++) {
6          tmp += *data_ptr++
7          if (tmp >= 256) {
8              tmp -= 255; // Wrap when overflow
9          }
10     }
11 }
```

Listing 2.1: C code snipped for calculating LIN frame checksum

The classic checksum method (For LIN 1.x slave nodes) does not include the ID as part as the checksum, where the enhanced method (For LIN 2.x slave nodes), does include the ID byte.

The use of LIN for ECU diagnostics

A variation of LIN called K-Line exists for diagnostic purposes only. This is a network which runs from the OBD-II port of the vehicle to a vehicles powertrain ECUs. This bus typically

runs at 10.4kbps and allows for the transmission of up to 255 bytes rather than the usual limit of 8 bytes when being used under the ISO14230-2 protocol (KWP2000).

As the bus is usually powered off during normal operation of the vehicle, it has to be woken up in one of 2 ways when a OBD-II adapter wants to send or receive data on the K-Line.

Five baud initialization

The Five baud initialization method is a slow wake-up method of the K-Line LIN network typically used for ISO9141-2 and works by the tester (The OBD-II adapter) sending the request ID at 5 bps. The requested ECU detects this slow initialization of the network and responds by waking up the network and sending its response ID back to the tester. Once the response ID has been received, the network is re-configured to run at 10.4 kbps. Once this is completed, no data can be sent or received for at least 300ms.

Fast initialization

The fast initialization method is another way to wake up a K-Line network, and is only used for ISO14230-2. With fast initialization, the tester sends a 25ms pulse on the K-Line, followed by the request ID at the networks usual bus speed (Typically 10.4kbps)

It should be noted that whilst the five baud initialization method and Fast initialization methods work very differently from one another, both can be achieved using the same physical hardware and transceivers.

ISO9141-2

ISO9141-2 ISO (1989) is the transport layer used for ISO9141-4 (OBD 2.3.1) when utilizing the K-Line diagnostic line of a vehicle. This configuration only utilizes the Five baud initialization sequence (Fast initialization is never used). This network configuration can run at multiple baud rates, ranging from 9.6kbps to 15.625kbps, however most common rates found are 9.6kbps and 10.4kbps. This configuration supports a maximum data size of up to 12 bytes, with an additional 1 byte used for checksum.

ISO14230-2

ISO14230-2 ISO (2000) is the transport layer used for ISO14230-4 (KWP2000 2.3.2) when utilizing the K-Line diagnostic line of a vehicle. This configuration of K-Line only runs at 10.4kbps (Unless its 5bps during the Five baud initialization sequence), and supports both Five baud initialization, or Fast initialization wake up methods. This configuration supports data sizes of up to 255 bytes, with an additional 1 byte for checksum.

With both of these network configurations, the target ECU on the K-Line will go back into a sleep state if no data is transferred in 5 seconds. Therefore, it is necessary for the tester to send periodic "Stay awake" messages on K-Line to keep the target ECU awake and in a diagnostic session.

2.2 Diagnostic adapter hardware and APIs

In this section, I will be discussing the various hardware and software used by most commercial diagnostic software to communicate with a vehicle using a specialized adapter.

2.2.1 Hardware APIs

In order to make it easier for a third party company such as Bosch to create an adapter for multiple diagnostic software suits for various OEMs, there are 2 main adapter APIs that most OEM diagnostic software supports. SAE J2534 (Passthru) and ISO 22900-2 (D-PDU). Each API has its own requirements and supports different physical communication protocols with a vehicle.

It should be noted that some OEMs use proprietary protocols for communicating with their own in-house diagnostic adapters, however in more recent times, such adapters are rare as most OEM's tend to support either Passthru or D-PDU, or sometimes both in their own diagnostic software suites, and offload the production cost of diagnostic adapters to third parties such as Bosch or XHorse.

SAE J2534

SAE J2534 (Passthru) Drew Technologies, Inc (2003) is a hardware API for diagnostic software to communicate with a supported adapter via a Windows DLL. It was originally created for Windows 2000 and Windows XP, however it still works on modern versions on Windows. A manufacturer of an adapter which supports the Passthru API can support any number of the following network layer protocols:

1. ISO 9141
2. ISO 14230-4
3. SAE J1850 41.6 KBPS PWM (Pulse width modulation)
4. SAE J1850 10.4 KBPS VPW (Variable pulse width)
5. CAN
6. ISO 15765-4 (ISO-TP)
7. SAE J2610 DaimelrChrysler SCI (Serial communication Interface)

Configuration data about each adapter and library which supports the Passthru API is stored in the Windows Registry. A typical registry entry for an adapter will contain the following information about the adapter:

- Hardware name
- Adapter vendor
- Supported protocols (From list above)
- Library path (Location to the Adapters DLL)

The J2534 API DLL must be compiled as a 32bit library DLL, meaning it is incompatible with 64bit software on modern systems. This is however not a problem as all diagnostic software is also compiled as a 32bit executable, in order to keep backwards compatibility with older systems.

ISO 22900-1 and ISO 22900-2

ISO 22900-1 (MVCI) and ISO 22900-2 (D-PDU API) Softing (2013) are both a hardware and software API for communicating with vehicles. Unlike SAE J2534, D-PDU is an entire protocol stack which includes diagnostic servers to do UDS (??), KWP2000 (??) and OBD-II (2.3.1), but for the context of this report, only ISO 22900-1 (hardware requirements) and ISO22900-2 (D-PDU API) will be discussed.

ISO 22900-1 mentions that an adapter that can be utilized by the D-PDU API can support any number of the following network layer protocols:

1. ISO 9141
2. ISO 14230-4
3. SAE J1850 41.6 KBPS PWM (Pulse width modulation)
4. SAE J1850 10.4 KBPS VPW (Variable pulse width)
5. CAN
6. CAN FD
7. ISO 15765-4 (ISO-TP)
8. Ethernet (DoIP)

Interestingly, ISO 22900-2 states that other third party adapter APIs can be utilized by D-PDU. This includes SAE J2534, however this is up to the application which implements the D-PDU API. This means that an application that uses the D-PDU API can work with an adapter that uses the SAE J2534 API, without actually having native support for the API itself, since D-PDU API server does the work with loading and calling the library for the J2534 adapter.

2.2.2 Hardware adapters

In this section, various diagnostic adapter hardware will be analysed and discussed. These adapters are used by a variety of different diagnostic tools/software.

Generic ELM327 BT adapters

ELM327 elmelectronics.com (2017) is a family of micro controllers which is used for communicating with vehicles using multiple network protocols. These controllers are most commonly found in cheap Bluetooth scan tools that only support OBD (ISO9141), however the micro controller itself can additionally support ISO 15765-4, ISO14230-4 and CAN, but these additional interfaces are rarely found in the cheaper scan tools as they utilize a 'cloned' version of the ELM327.

In order to interface with an ELM327, either USB or Bluetooth is utilized. The ELM327 receives and responds to a list of defined AT Commands. interestingly, since the AT Command set is publicly available and well documented, this has lead to open source projects which utilize micro controllers such as an ESP32 to emulate an ELM327. An example of such project can be found here [<https://github.com/collin80/AORET>]

The most common application which utilizes the ELM327 is Torque for Android (See 2.4.1).

A typical cost for an ELM327 Bluetooth adapter is about £10.0-20.0. However it is unknown if these adapter use a genuine ELM327 chipset or a cloned chipset.

SDConnect C4

The SDConnect C4 adapter is a diagnostic adapter which Daimler will ship with their diagnostic tool set (2.4.3) when provided to authorized workshops and dealers.

Due to this, not much is known about the cost of the adapter, however it is known it supports the D-PDU API, as well as Daimler's own proprietary protocol for talking to their in-house diagnostic adapters.

Whilst Daimler only provide the adapter to authorized workshops, there appears to be a huge amount of listings on ebay for this adapter, ranging in price from £500-£1000. However, it is apparent that some of these SDConnect listings might be cloned Chinese manufactured adapters, rather than a genuine adapter which has come from Daimler.

Bosch VCI

The Bosch VCI adapter is a series of diagnostic adapters which Bosch sells. For this section, the specifications of the Bosch MTS 6516 VCI adapter will be used.

This adapter can be used with both SAE J2534 and ISO 22900-1/2. It features 3 separate CAN channels (for CAN and ISO15765-4), 2 UART channels (For ISO9141 and ISO14230-2), 1 J1850 channel for either J1850VPW or J1850PWM. In order to connect to a PC, either WIFI or USB can be used with this adapter.

Cost of this adapter is unknown, however a similar used adapters by Bosch can be found on auction sites like Ebay for around £1000-£2000. So it can be concluded that these VCI adapters are very expensive, and out of the reach of the majority of consumers.

2.3 ECU Diagnostic protocols

In this section, the most common three diagnostic server protocols used in vehicles will be discussed (OBD-II, UDS and KWP2000).

To begin with, it should be noted that all these 3 services utilizes the same request and response message structure:

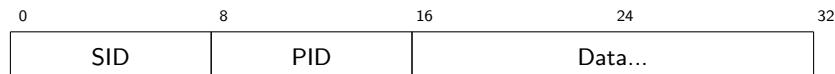


Figure 2.2: Data format of a service request with PID and data

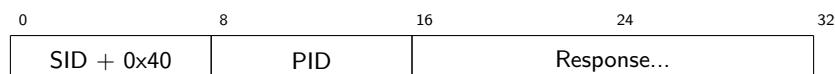


Figure 2.3: Data format of a positive ECU response

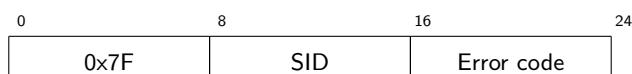


Figure 2.4: Data format of a negative ECU response

In the above figures, it should be noted that some services (Such as in OBD-II), do not require

a PID or Data, meaning only 1 byte of data (SID) is sent to the ECU. The entire length of the request and positive response messages can be as long as the underlying transport protocol. For example, ISO-TP can handle payloads of up to 4096 bytes, whereas ISO14230-2 can only support up to 255 bytes.

2.3.1 OBD-II

On-Board Diagnostics (OBD-II) is a relatively simple and read-only diagnostic protocol, and is a legal requirement on all vehicles since 2001. It is a standard way to communicate with the engine's ECU in a diagnostic session in order to read standard error codes, read sensor data from the ECU and carry out emissions tests. OBD-II can be used on any network layer protocol, but is mainly found to use ISO9141-2 on older vehicles, and ISO-TP on all cars found after 2008.

OBD-II have 9 pre-defined services, OEMs are free to choose which services their cars support, however most commonly services 01,02,03,04,09 are always implemented. It should be noted as well that OEMs are free to implement their own custom services beyond this range.

- 0x01 - Request current powertrain data
- 0x02 - Request powertrain freeze frame data
- 0x03 - Request emission-related diagnostic trouble codes (DTC)
- 0x04 - Clear/Reset emissions-related diagnostic information
- 0x05 - Request oxygen sensor monitoring test results
- 0x06 - Request ob-board monitoring test results for specific monitored systems
- 0x07 - Request emissions-related diagnostic trouble codes detected during the current or previous drive cycle
- 0x08 - Request control of on-board systems
- 0x09 - Request vehicle information
- 0x0A - Permanent DTCs

Every service (SID) has a child PID which will return which subfunctions are supported by the ECU for the SID. For example, taking a look at the response for SID 0x01, PID 0x00, which gets the supported PIDs for service 0x01 from 0x00 to 0x20:

```

1 REQUEST: 0x01 0x00
2 RESPONSE: 0x41 0x00 0xBE 0x1F 0xA8 0x13

```

Hexadecimal	B	E	1	F	A	8	1	3
Binary	1 0 1 1 1 1 1 0	0 0 0 1 1 1 1 1	1 0 1 1 1 1 1 1	0 1 0 1 0 1 0 1	0 1 0 1 0 1 0 1	0 0 0 0 0 0 0 0	0 0 0 1 0 1 0 1	0 0 0 1 0 1 0 1
PID supported?	Y N Y Y Y N N N	Y Y Y Y Y N Y N	Y Y Y Y Y N Y N	Y N Y N N N N N	Y N Y N N N N N	N N N N N N N N	N N N N N N N N	Y Y
PID ID (Hex)	01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20							

Table 2.2: Encoding example of supported PIDs for Service 01

From this, we can see that this engine ECU supports Service 01 PIDs 0x01, 0x03, 0x04, 0x05, 0x06, 0x07, 0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x13, 0x15, 0x1F and 0x20.

Service 0x01

Service 01 is used to retrieve live data metrics from the engine, PIDs 0x00, 0x20, 0x40, 0x60, 0x80, 0xA0, 0xC0 are special pids that are used to retrieve the next 32 supported PIDs, as shown in 2.2. A full list of PIDs are found at Wikipedia (2021).

Any response from Service 01 PIDs will result in 4 bytes or less being returned from the ECU, which are typically converted using either as enumerated values, or using mathematical formulas.

An example of how this works would be to look at decoding the Odometer reading over Service 01 (PID 0xA6):

```
1 REQUEST: 0x01 0xA6
2 RESPONSE: 0x41 0xA6 0x00 0xA0 0xC1 0x0F
```

The conversion formula is as follows:

$$\text{Odometer (Hm)} = \frac{A(2^{24}) + B(2^{16}) + C(2^8) + D}{10}$$

The *ABCD* encoding comes from the response bytes of the ECU, starting at the third byte in the response payload, since the first byte of a response is always a conformation of the Service ID being requested, and the second byte being the requested PID. The unit 'hm' implies Hectometre. 10 Hectometres = 1km

In this case, the bytes are converted to decimal as shown:

Byte	Hex	Decimal
A	0x00	0
B	0xA0	160
C	0xC1	193
D	0x0F	15

Applied to the bytes from the ECU, the conversion occurs:

$$\begin{aligned} &= \frac{0(2^{24}) + 160(2^{16}) + 193(2^8) + 15}{10} \\ &= \frac{0 + 10485760 + 49408 + 15}{10} \end{aligned}$$

To decimal numbers:

$$\begin{aligned} &= \frac{10535183}{10} \\ &= 1053518.3 \text{ hm} \implies 105351.83 \text{ km} \end{aligned}$$

Service 0x02

Service 02 works the same way as 01, however in addition to the PID, the Freeze frame number is also provided, and the ECU will return the sensor reading when the Freeze Frame occurred.

Service 0x03, 0x07 and 0xA

Services 03, 07 and 0A are all used to retrieve DTCs (Diagnostic trouble codes) that are stored on the ECU, and all function in the same manner. Service 03 is used to retrieve DTCs that are stored on the ECU (malfunctions that have occurred more than once and will result in the check engine light activating). Service 07 is used to retrieve DTCs that are pending (malfunctions that have been detected during the current/last drive cycle only, and haven't

since reoccurred). Service 0A is used to retrieve DTCs that are permanent (malfunctions that cannot be cleared with service 04 and will result in a failure in emissions test)

For all these services, no PID byte is given, and the ECU will return a list of error codes which are applicable to each service (2 bytes per DTC).

Service 0x04

Service 04 is used to clear any DTCs that are stored on the ECU. No PID bytes are required for this service, and the ECU does not respond back with any data. This service can only be used to clear stored and pending DTCs. Permanent DTCs cannot be cleared with this service, as they usually require the dealers specialist tool to clear.

Service 0x05 and 0x06

Services 05 and 06 work in identical ways, except service 05 is designed for non ISO-TP network transport layers, where as service 06 will only be executed on ISO-TP. Both services are used to monitor oxygen sensors in the engine which is a compulsory part of most emissions tests.

Service 0x08

Service 08 is used to control components in the vehicle when performing tests. For example, for the duration of some tests, the tester would tell the ECU with service 08 to not log any DTCs for the duration of the test. Once the test is over, the tester will use service 08 to tell the ECU to go back to its default state and monitor components and log DTCs.

Service 0x09

Service 09 is used to request vehicle information from the engine ECU. This can include data such as VIN (Vehicle Identification Number), Calibration ID, Calibration verification numbers (CVN), ECU Name and more. The majority of data returned in service 09 is returned as ASCII encoded strings.

Example of requesting the VIN:

```

1 REQUEST: 0x09 0x02
2 RESPONSE: 0x49 0x02 0x01 0x57 0x43 0x43 0x31 0x30 0x31 0x30 0x30 0x38 0x32 0x42
   0x32 0x37 0x33 0x39 0x32 0x32

```

Parsed VIN: **WCC1010082B273922**

2.3.2 KWP2000 and UDS

Keyword protocol 2000 (KWP2000 / ISO-14230-4) DaimlerChrysler (2002) is a diagnostic server protocol utilized by most vehicles from 2000, and was eventually replaced by UDS.

Unified diagnostic services (UDS) ISO (2006) is a diagnostic server protocol utilized by most vehicles from 2008. It is derived from KWP2000, thus shares a lot of similarities between KWP2000. Since most commands in UDS share the same functionality in KWP2000, refer to C.3 for a list of commands in both KWP2000 and UDS, as which ones are identical between protocols.

Both KWP2000 and UDS are both more advanced with OBD-II, relying on a diagnostic server on the ECU itself to communicate with diagnostic software over either UDS or KWP2000. The diagnostic server on the ECU can have different modes, which modify the behavior of the ECU. A default state is what the ECU is typically in on power on, then can

be modified to enter an extended diagnostic session, flash session (For firmware updates), or other protocol dependent modes. In these non-default modes, the ECU can behave unpredictably, or with reduced functionality. KWP2000 and UDS both have the ability to clear Permanent DTCs from an ECU, which OBD-II cannot do.

Both UDS and KWP2000 function over ISO-TP (2.1.2), however KWP2000 can also function on top of ISO14230-2 (2.1.3) and UDS can additionally function on DoIP (Diagnostic over IP). However, DoIP is typically only utilized during firmware updates of large ECUs such as infotainment systems in modern vehicles, which utilize a full Operating system. This dramatically reduces firmware update times from 10-12 hours (Over ISO-TP), to a few minutes (Over DoIP). This is due to DoIP using half duplex ethernet with a maximum bandwidth of 200Mbps, whilst ISO-TP is a lot slower, due to the underlying CAN network having a maximum speed of 500kbps, and also the ISO-TP protocol itself having an additional performance and bandwidth overhead.

2.4 Existing diagnostic software

In this section, the various diagnostic software available for vehicles will be discussed, ranging from basic OBD-II software to software made by the car manufacturers.

2.4.1 Torque for Android (Generic OBD)

Torque for android is an OBD-II diagnostic tool set designed for Android smartphones. It can utilize a wide range of adapters, but mainly ELM327 based bluetooth adapters, and comes in two versions. The 'Lite' edition is a free version and supports DTC Reading and clearing, and the ability to display a few Service 01 PIDs on graphs. The 'Pro' version costs £3.6 and supports everything that the Lite version does, but also includes all the Service 01 PIDs, GPS data logging and the ability to use external plugins.

2.4.2 Carly (Third party software)

Carly is a third party diagnostic mobile application. They provide specialist mobile apps for a lot of car manufactures, but also have a generic OBD-II mobile app for other car brands they don't natively support. The Carly application is free to use, but requires their own bluetooth OBD-II adapter must be used, which costs £64.90.

For supported car manufacturers, Carly supports advanced code reading (Via KWP2000 / UDS), as well as the ability to do SCN coding on select ECUs to enable or disable features on a vehicle. It also has an easy to use interface which can perform a 'used car buy check', which scans all the ECUs in a vehicle to ensure all the ECUs have the same milage reported, to verify that they have not been tampered with.

2.4.3 Xentry (Dealer software)

Xentry diagnostic suite is Daimler's in-house diagnostic software. It consists of two main programs, DAS (Diagnostic assist service), which is utilized on pre 2008 vehicles, and Xentry, which is utilized on all vehicles manufactured after 2008.

In terms of data formats, DAS utilizes the older CBF File format, whilst Xentry utilizes a newer binary data format called SMR-D.

Both pieces of software work in nearly the same way. When the software is connected to a vehicle via the means of a valid diagnostic adapter, the software first attempts to retrieve the vehicles VIN number, which is then used to decode what ECUs the vehicle has. With that

knowledge, the diagnostic application can then scan all the ECUs within the vehicle using the correct communication protocols and typically using KWP2000 or UDS diagnostic servers. This can be viewed in A.1.

Once the application has established contact with all the ECUs within the vehicle, the tester (The engineer who uses the software) is then free to probe each individual ECU in the vehicle in order to locate a problem (A.2). Each ECU in the vehicle has a list of adaptations and tests that can be carried out in order to test a potential problematic component. The tester is guided throughout the tests by the diagnostic software as to what to do next, for instance turning on/off the vehicle. Some tests can even be carried out by the ECU fully autonomously, with the results being displayed to the user once the test has completed.

Each ECU also has its own view dedicated to displaying a list of error codes stored on the ECU (DTCs), as well as displaying what the ECU was doing at the time of the error (Freeze Frame data). The tester can view an error as well as a full description of the error.

2.5 Open Diagnostics eXchange (ODX)

The Open Diagnostic eXchange format (ODX) is a way for vehicle manufacturers to define an entire specification for an ECU or vehicle, which then gets distributed to both ECU manufacturers, such as Bosch or Delphi and the organizations which create commercial diagnostic software for both the OEM and third parties.

The ODX specification consists of six main categories:

ODX-D	Diagnostic data
ODX-C	Communication parameters
ODX-V	Vehicle topology
ODX-F	Flash data container
ODX-E	ECU Coding data
ODX-FD	Mapping to functions

There are very few free tools which can read and interpret ODX files. Instead, companies such as Vector and Softing have commercial licensed tools that can generate ODX data files, as well other tools that can be used to test an ECU's implementation of the ODX data.

Diagnostic software typically utilizes data from all the above ODX categories, which the OEM or diagnostic software creator will then use to compile custom binary files for their own tools based on the data from ODX. This is done so that the raw ODX data is no longer in a raw format, which can be easily understood by individuals, and also saves a lot of space compared to using the raw XML based ODX containers. An example of this proprietary binary format that will be utilized in this report is Daimler's CBF file format.

2.6 The OBD-II port

The OBD-II diagnostic port is a standard connector found on all commercial vehicles since 1996. The connector is very recognizable as a female J1962 plug with 16 (2 rows of 8) pins.

The OBD-II standard dictates that the female plug must be within 2 feet of the steering wheel of the vehicle, and on the inside of the vehicle. The Standard pin out of the J1962 connector is as follows:

1	OEM dependant	9	OEM dependant
2	J1850 bus positive	10	J1850 bus negative
3	OEM dependant	11	OEM dependant
4	Chassis ground	12	Not connected
5	Signal ground	13	Not connected
6	CAN High	14	CAN Low
7	K-Line for ISO1941 and ISO14230	15	L-Line for ISO1941 and ISO14230
8	OEM dependant	16	Battery voltage

However, the pin layout of this connector is modified if the plug is compliant with 13400-2, or DoIP (Diagnostic over IP). This is a recent addition, and only a few OEM's currently use it, however it allows for a DoIP compliant diagnostics adapter to connect to the vehicles internal Ethernet network for performing operations such as firmware updates.

1	OEM dependant	9	OEM dependant
2	J1850 bus positive	10	J1850 bus negative
3	Ethernet TX+	11	Ethernet TX-
4	Chassis ground	12	Ethernet RX+
5	Signal ground	13	Ethernet RX-
6	CAN High	14	CAN Low
7	K-Line for ISO1941 and ISO14230	15	L-Line for ISO1941 and ISO14230
8	Ethernet wake up	16	Battery voltage

2.7 Comparisons to the proposed project

As mentioned in (2.2) and (2.4), there are multiple existing solutions for diagnostic adapters and diagnostic software for vehicles. In this section, a comparison will be made between the various adapters and existing software, also including the proposed solution for each category.

2.7.1 Hardware adapters

Feature	Bosch VCI	Elm327	Proposed
Cost	£500	£20	£60
OS Support	win32	Any	win32, OSX, Linux
J2534 API	Yes	No	Yes
D-PDU API	Yes	No	No
CAN	Yes	Yes	Yes
ISO-TP	Yes	Yes	Yes
ISO14230-2	Yes	Yes	Yes
ISO9141	Yes	Yes	Yes
Connection method	USB/Ethernet	BT/USB	USB

Table 2.3: Comparison between existing adapters and proposed solution

As seen in the above table, the proposed adapter solution will be the only one which can support the J2534 API on all desktop operating systems. One thing to note about the ELM327 column. As discussed in (2.2.2), there can be many clone adapters, which won't support all the transport protocols such as ISO-TP, instead, only supporting ISO9141 so that they can be used with mobile applications such as Torque.

2.7.2 Diagnostic software

Feature	Xentry (Dealer)	Carly (Third party)	Torque (OBD)	Proposed
Cost	Unknown	£69 (With adapter)	£3	Free
OS Support	Win32	Mobile	Mobile	Win32, OSX, Linux
J2534 API	Yes	No	No	Yes
SocketCAN	No	No	No	Future
D-PDU API	Yes	No	No	Future
OBD-II support	No	Yes	Yes	Yes
KWP2000 support	Yes	Yes	No	Yes
UDS support	Yes	Yes	No	Yes
Clear permanent DTCs	Yes	Yes	No	Yes
Communicate with all ECUs in a vehicle	Yes	Yes	No	Yes
Live data reading	Yes	Yes	Yes (OBD only)	Yes
ECU test execution	Yes	No	No	Yes

Table 2.4: Comparison between existing diagnostic software and proposed solution

As seen in this table, the proposed solution will be the only one which can support Linux and OSX. Also, SocketCAN and D-PDU are proposed as future additions to the application, but for the current scope, they will not be included due to time constraints.

Chapter 3

Methodology

This section will cover the design and implementation of all 3 parts of the project, referencing content discussed in the literature review. Each part is broken down into its design, implementation and justification.

3.1 Test setup

Prior to designing and implementing any parts of the solution, it was clear that a permanent test setup would be needed to test with, in order to avoid the risk of accidentally damaging something in a real vehicle. Therefore, it was decided to use the following test ECUs on a desk to form a simulated setup of a real vehicle. In this case, the following ECUs were used which can replicate part of a Mercedes W203.

ECU Name	Description	CAN C	CAN B	K-Line
IC203	Instrument cluster	Yes	Yes	No
CRD	Engine ECU	Yes	No	Yes
EGS52	Transmission controller	Yes	No	Yes

Table 3.1: ECU List in the desk test setup

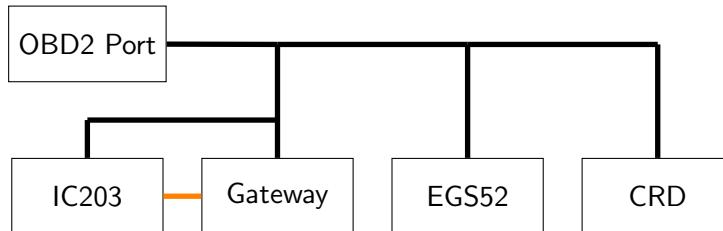


Figure 3.1: Mock-up of how the ECUs are connected in the test setup

In this test setup, the 'Gateway' is simply An arduino with 2 CAN interfaces on it. Its job is to relay CAN Frames to and from the CAN C Network (Black) and the CAN B network (Orange). This is because for diagnostics, the cluster only responds if the diagnostic packets are sent over CAN B, and not CAN C. In a real vehicle, this functionality is usually handled by the ignition switch ECU which acts as a CAN Gateway. However, this could not be purchased for this setup due to its cost.

3.2 Rust

As part of this project, the Rust programming language was utilized rather than other languages such as C++ or Java for the majority of the development process, however C++ had to be used for the Passthru adapter firmware [3.4], due to limitations with the standard Arduino libraries. Rust has some unique advantages compared to other languages, to name a few:

- Thread and memory safety guaranteed at compile time
- Easy dependency management with Cargo
- Easy cross-compilation support
- Bare-metal performance (Unlike JVM based languages)

However, at the time of writing this, Rust does have bindings for platform GUI library's, such as QT, GTK and WINAPI, but not have a mature cross-platform GUI library, which would be required for the Diagnostic application, as will be mentioned in section 3.5. However, an experimental cross-platform library called Iced [<https://github.com/hecrj/iced>] does exist. This GUI library supports all three major operating systems (Linux, Windows and MacOS), as well as the Web for WASM build targets. Everything in this library is drawn to the screen using either Vulkan, OpenGL, Metal or DX11 depending on the platform.

Iced is inspired by the Elm Architecture, meaning that the GUI is built in pure code, rather than having external XML files to describe the UI. Also due to it being inspired by Elm, the GUI architecture is split into four main components. Application state, message events within the UI, view logic and update logic.

3.3 JSON schema creation and CBF parsing

As mentioned in 1, this section will cover the creation of OpenVehicleDiag's JSON Schema which is used as a easy-to-read replacement for ODX-D [2.5].

3.3.1 JSON structure

Since Rust has an excellent JSON Serialization with Serde [<https://serde.rs/>], the following UML class diagrams will show the design of the underlying structs used in the application. This then gets serialized into JSON by Serde. A full JSON Schema is available in section E, as well a full example of the JSON in section C.4.

Since it is almost guaranteed that the data will be similar, or at least comparable to what is found in ODX-D [2.5], [emotive de (2014b)] and [emotive de (2014a)] were used as references when designing the JSON structure, which heavily document the various data types and presentation formats found in ODX files.

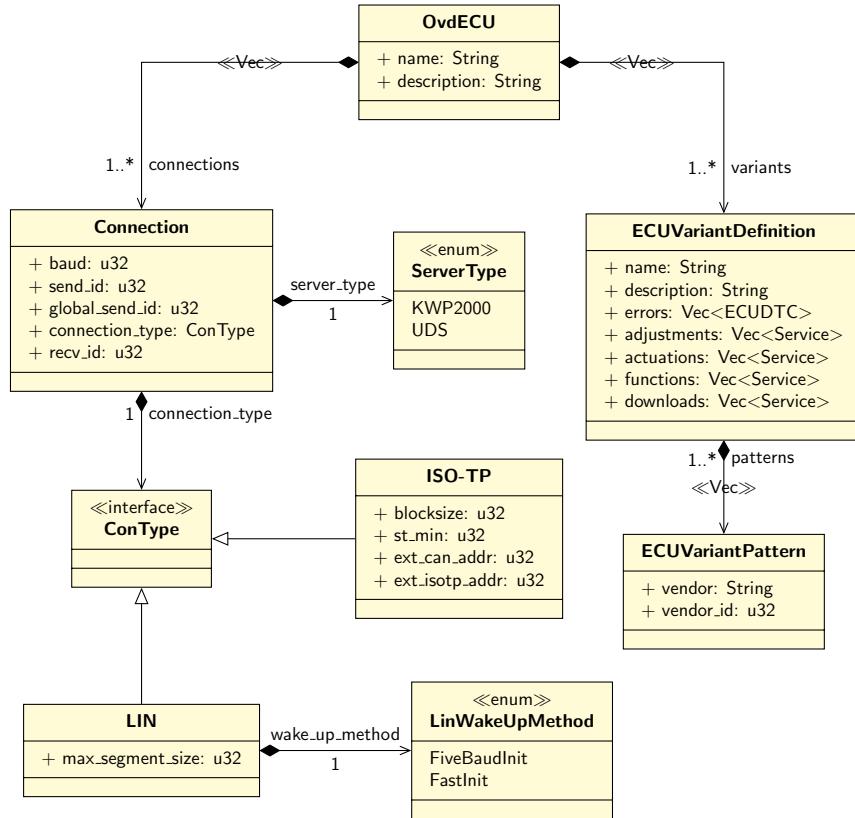


Figure 3.2: UML representation of ovdECU Root object, ECUVariantDefinition and Connection properties

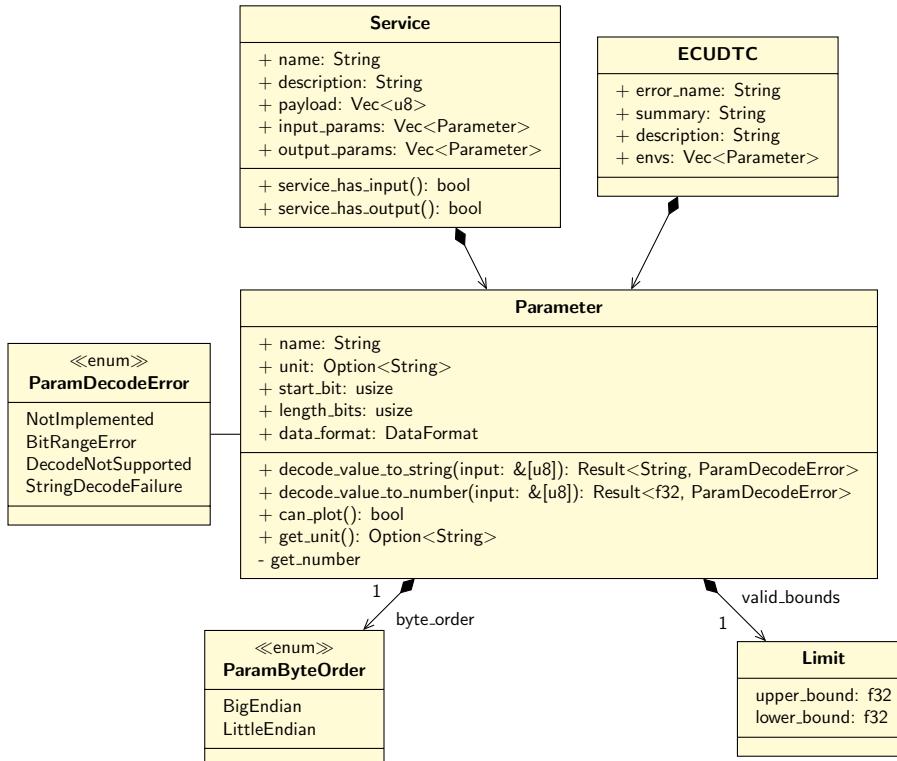


Figure 3.3: UML representation of JSON Service and ECUDTC

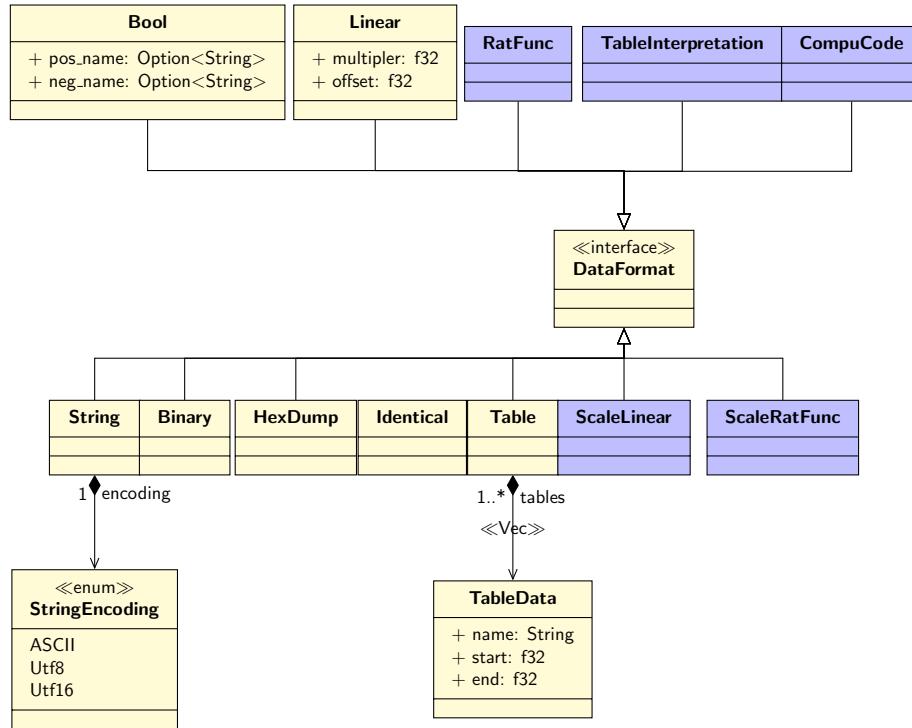


Figure 3.4: UML representation of JSON DataFormat

Figure 3.2 shows the root object UML representation, as well as connection properties and the main object for storing a ECU Variant object.

OvdECU is the root structure of the JSON. This contains the name of the ECU, a description of the ECU, connection properties of the ECU, and all ECU Variants. A ECU must have at least 1 Connection method and at least VariantDefinition.

Connection is used to define what transport and diagnostic protocols are used when communicating with the ECU. This contains data such as the bus speed (baud) of the connection method as well as various ID's to use whilst in a diagnostic session. An ECU can have multiple connection methods, such as KWP2000 over ISO-TP (CAN), or KWP2000 over K-Line (ISO 14230-2). In either configuration, the ECU's addresses will differ. **send_id** is used to identify which ID on either network type the tester should send diagnostic messages on. **recv_id** denotes which ID the ECU will respond on, and the tester will listen for data with this ID. **global_send_id** is an optional entry, and is only used for some ECUs. This value is required in situations where the TesterPresent message is to be broadcast on a separate address to the diagnostic request message (Normally they are the same address). In the case of my test vehicle, this is used for all interior CAN devices located on CAN-B.

ConType represents an interface for each physical network connection method to inherit, with various attributes related to the network setup. For instance, ISO-TP configuration requires attributes attaining to block size, default separation time, and extended addressing, whilst the LIN connection (K-Line) requires a maximum packet size that the ECU supports, as well as which wake up method to use when activating the K-Line network.

ECUVariantDefinition contains the definition for each ECU Variant. A ECU Variant is an equivalent to a software version on a desktop PC. An ECU can receive updates over time (via firmware flashes). With each firmware update, diagnostic routines can be altered, as well as lists of errors, or even in some cases, the description of certain errors changes from variant to variant. Each ECU Variant can be implemented by multiple hardware manufacturers

such as Bosch or Delphi. Because of this, the ECUVariantPattern class is required here. This small class contains the name of the manufacture (Vendor), as well as a defined vendor ID. This vendor ID can be retrieved over both KWP2000 and UDS, and is unique to the software version and ECU manufacturer.

Figure 3.3 shows how both Service and ECUDTC both use the Parameter object.

Service represents an IO function that can be executed on the ECU. Each function requires a name, description of what the function does, a raw payload to send to the ECU, and a list of input and/or output parameters. Input parameters modify the payload that is sent to the ECU, whilst output parameters are used to convert the ECUs response message into something that can be easily interpreted by an individual.

ECUDTC represents a Diagnostic trouble code (DTC) that can be stored on the ECU. Each DTC has a unique name, such as 'P202A', a summary of what the error means, and a more detailed description string of what the error means. the envs array is a list of parameters that are used when querying freeze frame data about the DTC, which is then interpreted using the list of parameters to decode the freeze frame data into something that is human readable. Freeze frame data can be acquired with both KWP2000 and UDS.

Parameter contains data which has been interpreted from the ODX-D data structure [emotive de (2014a)]. Each parameter contains a name which denotes what the parameter means, as well as an optional unit, which can be used to denote the unit of the parameter, such as $^{\circ}\text{C}$ or Engine RPM. start_bit denotes where in the payload the parameter starts at, and length_bits denotes how big the parameter is in bits. This is because the ECU response message can contain values that are tightly packed, and at odd bit offsets. data_format denotes which Data format interpreter to use for the raw data within the given bit range. Lastly, valid_bounds is an optional class which can be specified. If it is declared in a Parameter which is stored in a parent services' input_parameters section, then it is used to check the user input is within range. If it is stored in a parameter which lies within a parent services' output_parameters section, then it used to set the upper and lower bounds for graphing. Note that valid_bounds is ignored if it is specified in a parameter which cannot be converted to a number data type.

Figure 3.4 Shows the classes which inherit the DataFormat interface. Classes that appear in blue exist in code as a placeholder, and do not do anything at this time. They are derived from the ODX-D specification.

Bool data format is used when the parent parameter shall be interpreted as a boolean. This class has two optional parameters, which can be used to override the default 'True', 'False' interpretation of booleans. This data type can be used for graphing.

Linear format is used when a raw value is to be manipulated using a simple $y = mx + c$ equation. The input value is x , whilst 'multiplier' represents m , and 'offset' represents c . This data format can be used for graphing.

String format is used when the parent parameters byte range is to be interpreted as a raw string. This data format has a parameter (encoding), which denotes what text encoding to use when encoding and decoding the string. This can either be ASCII, Utf8 or Utf16. This data type cannot be used for graphing.

Binary format is used when the parent parameter shall be interpreted as a binary string, such as '0b00110011'. It should be noted that this data type is not in the ODX-D specification, but was added at a later date due to finding that Daimler seemed to use Enum tables to represent binary strings in their CBF files, meaning that an 8 byte parameter would have

256 enum entries! This data type cannot be graphed.

HexDump format simply returns a formatted string of the bytes within the parent parameters bit range.

Identical format is used for when the raw bytes of the parent parameter is to be interpreted as a 32bit integer and returned as is without any manipulation such as with the Linear data type. The parent parameter's byte_order will specify how to interpret the bytes which make up the integer. This data type can be graphed.

Table format is used for representing enum values. This means that a number value or range is mapped to a string. This data type takes a compulsory list of more than 1 TableData entries. Each entry consists of a minimum (inclusive) and maximum (inclusive) value, as well as the definition string. A number range is used here rather than a one to one mapping since this is what the ODX-D specification states. This data type cannot be graphed.

3.3.2 Code implementation of the JSON Schema

To translate the UML diagrams to code was a very simple process. Below is the code implementation of the OvdECU Struct:

```

1 #[derive(Debug, Clone, Serialize, Deserialize)]
2 pub struct OvdECU {
3     pub name: String,
4     pub description: String,
5     pub variants: Vec<ECUVariantDefinition>,
6     pub connections: Vec<Connection>
7 }
```

On line 1, the `#[derive]` attribute tells the compiler to implement traits for the Struct at compile time. `Serialize`, `Deserialize` are traits that come from the Serde library, and will enable this Struct to be serialized and deserialized to and from JSON without any additional code needing to be written.

Looking at the Service Struct, the implementation in code is a bit more complicated:

```

1 #[serde_as]
2 #[derive(Debug, Clone, Serialize, Deserialize, PartialEq)]
3 pub struct Service {
4     pub name: String,
5     pub description: String,
6     #[serde_as(as = "serde_with::hex::Hex<serde_with::formats::Uppercase>")]
7     pub payload: Vec<u8>,
8     #[serde(skip_serializing_if = "Vec::is_empty")]
9     #[serde(default = "Vec::new")]
10    pub input_params: Vec<Parameter>,
11    #[serde(skip_serializing_if = "Vec::is_empty")]
12    #[serde(default = "Vec::new")]
13    pub output_params: Vec<Parameter>
14 }
```

In this structure, there are few more additional tags that are used by Serde. `skip_serializing_if` is used to tell Serde to skip Serialization if a condition is met for the value. In this case, if the Vector is empty, Serialization is skipped in order to reduce the Serialized file size. `default` tells Serde what to do if the value is not found in JSON when deserializing. In this case, replace the value with an empty vector. Lastly, the `serde_as` on line 6 tells Serde to serialize the payload as an upper case Hex String. This because by default Serde will serialize the payload as a list of numbers, which get re-interpreted as bytes during deserialization. This was not optimal as both KWP2000 and UDS use Hex bytes in documentation rather than numbers. Therefore, this tells Serde to strictly serialize the payload as a hex string, which is more easily understood by people reading the file.

Implementing Parameter functions

As shown in figure 3.3, the Parameter object has functions that are used to decode the parameter to either string or integer. This is then used by any application which utilizes the JSON Schema to decode an ECU byte stream. The following Matrix denotes which function is supported by which internal DataFormat (Figure 3.4):

Data format name	Format as string?	Format as number?
HexDump	Yes	No
Binary	Yes	No
String	Yes	No
Bool	Yes	Yes
Table	Yes	No
Identical	Yes	Yes
Linear	Yes	Yes

Table 3.2: Supported format options for DataFormat

If `decode_value_to_number()` is called on a Parameter which cannot be encoded as a number (such as String), then an Error of type `DecodeNotSupported` is returned from the function. This table also shows that `decode_value_to_string()` is supported by every data format type.

Both functions work in the same manner. They take a byte stream, which is the ECU response message to a command, and extract bytes within a specific byte range, then process the bytes. In the event that the parameter bit range lies outside the ECU response message (For example, receiving a message of length 120 bits, but the parameter specified a start bit of 150), the an Error of type `BitRangeError` is returned.

If either decoder expects its data type to be a number, then `get_number()` is called, which can at most return a 32bit long number. This seemingly arbitrary restriction of 32bits is due to after extensive testing, all ECUs seem to only handle 32bit numbers and nothing more. This restriction can be increased to 64bits with a simple addition to the codebase.

```

1 fn get_number(&self, resp: &[u8]) -> std::result::Result<u32, ParamDecodeError> {
2     if self.length_bits <= 32 {
3         let result = std::panic::catch_unwind(|| {
4             if self.length_bits <= 8 {
5                 resp.get_bits(self.start_bit..self.start_bit+self.length_bits) as u32
6             } else {
7                 let mut res = 0;
8                 let mut buf: Vec<u8> = Vec::new();
9                 let mut start = self.start_bit;
10                while start < self.length_bits + self.start_bit {
11                    let max_read = min(self.start_bit + self.length_bits, start + 8);
12                    buf.push(resp.get_bits(start..max_read));
13                    start += 8;
14                }
15
16                if buf.len() > 4 {
17                    panic!("Number too big!") // Cannot handle more than 32bits atm
18                } else {
19                    if buf.len() >= 4 {
20                        res = match self.byte_order {
21                            ParamByteOrder::BigEndian => BigEndian::read_u32(&buf),
22                            ParamByteOrder::LittleEndian => LittleEndian::read_u32(&buf)
23                        }
24                    } else if buf.len() >= 2 {
25                        res = match self.byte_order {
26                            ParamByteOrder::BigEndian => BigEndian::read_u16(&buf) as u32,
27                            ParamByteOrder::LittleEndian => LittleEndian::read_u16(&buf) as u32
28                        }
29                    }
30                res as u32
31            }
32        });
33    });
34}

```

```

35     match result {
36       Ok(r) => Ok(r as u32),
37       Err(_) => Err(ParamDecodeError::BitRangeError)
38     }
39   } else {
40     Err(ParamDecodeError::BitRangeError)
41   }
42 }
```

This function essentially extracts a 1-32bit long number from the input byte stream, using the Parameters start bit and length bits values to work out where to extract the number in the input array. As seen in this code, it also takes into account the Endianness of the expected value.

When decoding a value to String, each DataFormat has its own parser function that runs on the input byte stream. Below is a couple examples:

```

1 DataFormat::Bool { pos_name, neg_name } => {
2   return match self.get_number(input)? {
3     0 => Ok(neg_name.clone().unwrap_or("False".into())),
4     _ => Ok(pos_name.clone().unwrap_or("True".into()))
5   }
6 }
7
7 DataFormat::HexDump => {
8   let start_byte = self.start_bit/8;
9   let end_byte = (self.start_bit+&self.length_bits)/8;
10  return Ok(format!("{:02X?}", &input[start_byte..min(end_byte, input.len())]))
11 }
12
13 DataFormat::Linear { multiplier, offset } => {
14   let res = self.get_number(input)? as f32;
15   result.push_str(format!("{}", (res*multiplier) + offset).as_str())
16 },
17 }
```

As seen here, both Bool and HexDump return early with their formatted Strings, whilst Linear doesn't return early, this is because it is a number and therefore might have a unit associated with it. Further down the decoder there is a check for this, and if a unit exists, it is appended to the 'result' string, before being returned.

3.3.3 Parsing Daimler CBF Files to JSON

For this section, CaesarSuite [<https://github.com/jglim/CaesarSuite/>] was heavily used as a reference on how to extract data from Daimlers CBF Files. CaesarSuites creator (JinGen Lim) has spent countless hours reverse engineering `c32s.dll`, which is a DLL file which ships with Daimler's diagnostic tool set, and is responsible for loading CBF files amongst other things.

After re-writing the majority of the CBF Parser code found in CaesarSuite to Rust (CBF-Parser), the codebase now had to convert between Daimler's CBF representation structures to the JSON Schema structure. This proved to be a challenge since the data structures are completely different. Below is the UML representation of a CBF File, without any field values:

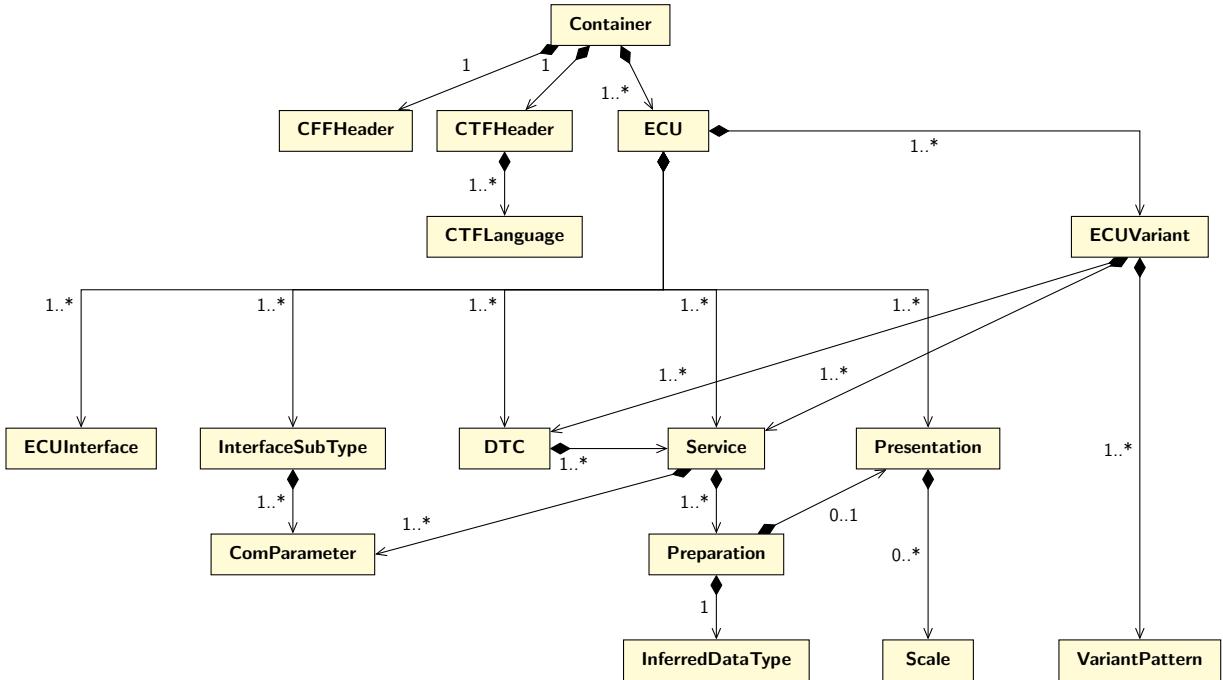


Figure 3.5: Simplified UML representation of the data structure in a CBF File

As shown in figure 3.5, a CBF File has a completely different data structure when compared to the JSON Schema. To solve this, a parser script was written to process the root Container and convert this data structure to the JSON structures. This will be very briefly touched upon here, since the code to convert to the JSON schema is far too long.

As an example, here is the code which converts the ComParameter's found in InterfaceSubType to a Connection property for the JSON:

```

1 let mut connections = Vec::new();
2 for x in e.interface_sub_types.iter() {
3     let connection = if x.comm_params.iter().any(|x| x.param_name == "CP_REQUEST_CANIDENTIFIER") { // ISOTP
4         Connection {
5             baud: x.get_cp_by_name("CP_BAUDRATE").expect("No CAN Baudrate on interface!?" ),
6             send_id: x.get_cp_by_name("CP_REQUEST_CANIDENTIFIER").expect("No CAN ReqID on interface!?" ),
7             recv_id: x.get_cp_by_name("CP_RESPONSE_CANIDENTIFIER").expect("No CAN RespID on interface!?" ),
8             global_send_id: x.get_cp_by_name("CP_GLOBAL_REQUEST_CANIDENTIFIER"),
9             connection_type: ContType::ISOTP {
10                 blocksize: 8, // Some reason MB always uses 8
11                 st_min: x.get_cp_by_name("CP_STMIN_SUG").unwrap_or(20), // Seems default for MB
12                 ext_isotp_addr: false, // MB never use extended ISO-TP addressing
13                 // Check CAN ID to see if the CAN network itself is extended
14                 ext_can_addr: x.get_cp_by_name("CP_REQUEST_CANIDENTIFIER").unwrap() > 0x7FF
15                 || x.get_cp_by_name("CP_RESPONSE_CANIDENTIFIER").unwrap() > 0x7FF
16             },
17             server_type: if x.qualifier.contains("UDS") { // Interface type is in qualifier name for ISO-TP
18                 ServerType::UDS
19             } else {
20                 ServerType::KWP2000
21             }
22         }
23     } // Assume LIN (ISO14230-2)
24     Connection {
25         baud: 10400, // Always for ISO14230-2 with MB
26         send_id: x.get_cp_by_name("CP_REQTARGETBYTE").expect("No LIN Request ID on interface!?" ),
27         recv_id: x.get_cp_by_name("CP_RESPONSEMASTER").expect("No LIN Response ID on interface!?" ),
28         global_send_id: x.get_cp_by_name("CP_TESTERPRESENTADDRESS"),
29         connection_type: ContType::LIN {
30             max_segment_size: x.get_cp_by_name("CP_SEGMENTSIZE").unwrap_or(254), // Default for ISO14230-2
31             wake_up_method: LinWakeUpType::FiveBaudInit, // MB always uses this with ISO14230-2
32         },
33         server_type: ServerType::KWP2000 // Always with LIN
34     }
35 };
36 connections.push(connection);
37 }
  
```

What this code does is iterate over each of the CBF's InterfaceSubTypes, which represents

a connection method for the ECU, and try to work out what kind of connection it is based on the ComParameters found in the InterfaceSubType, then mapping the ComParameters to data in the Connection object for the JSON Schema.

The name of the ComParameters (EG: CP_BAUDRATE) are all found within the ODX specification, so it is simple to know what they do. Also to note is that especially for a LIN connection type, there is a lot of assumptions about what Mercedes utilizes. This was done by looking at how Daimler's own software tries to initialize contact with an ECU using ISO14230-2 for over 50 different ECUs in various cars that supported ISO1423-4.

CBF parser optimizations

After processing many CBF files, it was noticed that CBF files seem to contain many Services with the same request payload, and only the output_params would vary. This significantly increased the size of the output JSON file. An example of this behavior is shown below from parsing the CRD ECU's CBF file:

```

1 "downloads": [
2   ...
3   {
4     "name": "DT_IOC0_Dpf_spy_cata_temp_nvv_5",
5     "description": "cata temperature in regeneration [5]]: 610 C",
6     "payload": "30C001",
7     "output_params": [
8       {
9         "name": "CRD.PRES_Conversion_52_Status",
10        "unit": "",
11        "start_bit": 2024,
12        "length_bits": 16,
13        "byte_order": "BigEndian",
14        "data_format": {
15          "Linear": {
16            "multiplier": 1.0,
17            "offset": 0.0
18          }
19        }
20      }]
21    },
22  {
23    "name": "DT_IOC0_Dpf_spy_cata_temp_nvv_6",
24    "description": "cata temperature in regeneration [6]]: 620 C",
25    "payload": "30C001",
26    "output_params": [
27      {
28        "name": "CRD.PRES_Conversion_52_Status",
29        "unit": "",
30        "start_bit": 2040,
31        "length_bits": 16,
32        "byte_order": "BigEndian",
33        "data_format": {
34          "Linear": {
35            "multiplier": 1.0,
36            "offset": 0.0
37          }
38        }
39      }]
40    },
41  ...
42 ]

```

This is just one example. The CRD ECU in total has over 3500 services that look like this. By grouping services' output_params together based on the request payload, it was possible to significantly reduce the file size of the output JSON. Below is the same service's grouped together:

```

1 "downloads": [
2   ...
3   {
4     "name": "DT_30_C0",
5     "description": "Data download 30 C0",
6     "payload": "30C001",
7     "output_params": [
8       ...
9       {
10         "name": "counter for the conditions at the end of the regeneration [0]",
11         "unit": "",
12         "start_bit": 56,
13         "length_bits": 16,
14         "byte_order": "BigEndian",
15         "data_format": {

```

```

16         "Linear": {
17             "multiplier": 1.0,
18             "offset": 0.0
19         }
20     },
21 },
22 {
23     "name": "counter for the conditions at the end of the regeneration [1]",
24     "unit": "",
25     "start_bit": 72,
26     "length_bits": 16,
27     "byte_order": "BigEndian",
28     "data_format": {
29         "Linear": {
30             "multiplier": 1.0,
31             "offset": 0.0
32         }
33     },
34 },
35 ...
36 },
37 ...
38 ]

```

The only down side to this approach is that the name and description of the parent service is now a generic name, since the name is now no longer known. However, this is not a problem as shown in section 3.5, the GUI for interfacing with this generated JSON will search both input and output parameters for a user's search terms, meaning that the parent service name doesn't have to be descriptive. Also, by doing this the file size of the output JSON shrunk by approximately 15%, which in turn decreases parse times and load times in OpenVehicleDiag.

3.4 Cross platform Passthru adapter

The Macchina M2 Under-the-dash adapter was chosen for this task, primarily due to the following reasons:

- Open source hardware schematics available on Github
- Excellent support with library authors
- Built on a mature platform based on hardware found in the Arduino Due
- Can be easily programmed using existing tools designed for Arduino based devices such as Arduino IDE or VSCode
- Native USB support - Data transfers over a USB cable can occur at much higher speeds when compared to other Arduino based solutions that require a USB to Serial converter chip. This is a feature of the AT91SAM3X8E CPU rather than an explicit feature of this OBD-II adapter.
- Supports all of the necessary interfaces for the J2534 API (CAN, LIN, J1850, SCI)
- Reasonably priced at \$99 - Which is a lot cheaper compared to a lot of alternative J2534 devices as discussed in section 2.2



Figure 3.6: Macchina's M2 Under the dash OBD-II module

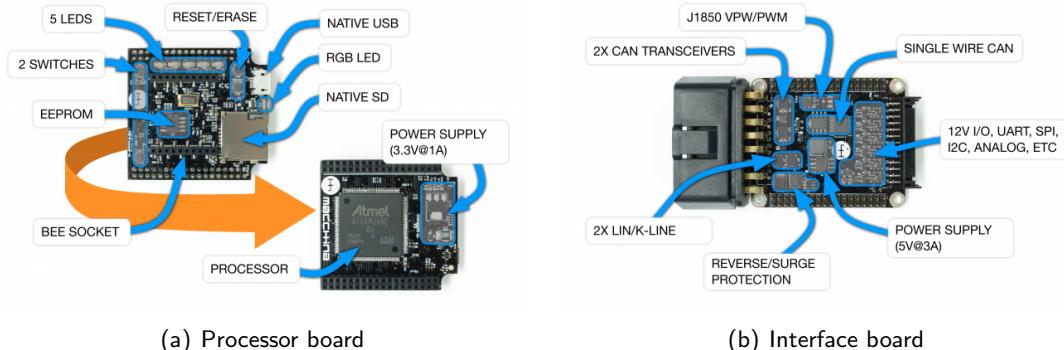
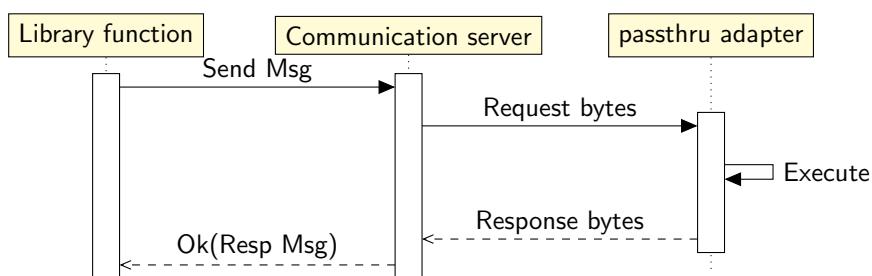


Figure 3.7: Macchina M2 board layouts

The SAE J2534 reference manual [Drew Technologies, Inc (2003)] was used as a reference for the API implementation. Specifically, this will be an implementation of V04.04 of the Passthru API¹. Due to the short time constraints of this project, Only CAN and ISO-TP will be implemented.

3.4.1 Architecture

In order to reliably allow the J2534 library and adapter to communicate, the architecture of the communication between the adapter and J2534 library will revolve around Request and Response messages. This can be executed without overly complex code since the J2534 API states the the API is designed to be single threaded, meaning that any application that utilizes the J2534 API will not be making asynchronous calls to the library.



The above figure shows how communication between the library and adapter itself will be handled. The library will contain various functions for tasks like channel creation or transmitting data, and they will construct Payloads to send to the adapter. These payloads are forwarded to a communication server which is part of the library, and sends the payload as a byte stream to the adapter. The adapter will execute what the payload specifies, and then return a response message to indicate if the action completed successfully, or failed (Which will contain the error message). This response is then forwarded back to the caller library function.

Since the M2 has 5 LED's and an additional RGB LED, it was decided to utilize these LEDs so the user knows what is going on without the need to check log files:

¹List of Passthru API functions: C.1

LED Name	Colour	Usage
DS6	Green	Indicate an application is currently utilizing the adapter.
DS5	Yellow	Indicates a J1850 channel is active.
DS4	Yellow	Indicates a SCI, ISO9141 or ISO14230 channel is active.
DS3	Yellow	Indicates a CAN or ISO-TP channel is active.
DS2	Red	On when adapter first boots, indicates firmware is ready, but no application is utilizing it.
DS7_RED	RGB Red	On when adapter receives data from PC
DS7_GREEN	RGB Green	Not used
DS7_BLUE	RGB Blue	On when adapter transmits data to PC

3.4.2 Creating the driver in Rust

Firstly, the J2534 library had to be ported to Rust from C++. This means converting the definitions found in the J2534 API [Comer352L (2019)], to something that Rust's type system can work with more easily. As part of this conversion process. Most of the define's found in the J2534 header file were converted to enum decelerations in Rust.

Below is an example of the protocols' section of J2534.h:

```

1 // Protocols:
2 #define J1850VPW 0x01
3 #define J1850PWM 0x02
4 #define ISO9141 0x03
5 #define ISO14230 0x04
6 #define CAN 0x05
7 #define ISO15765 0x06
8 #define SCI_A_ENGINE 0x07
9 #define SCI_A_TRANS 0x08
10 #define SCI_B_ENGINE 0x09
11 #define SCI_B_TRANS 0x0A

```

and here is how that is translated into an enum structure in Rust:

```

1 #[repr(u32)]
2 #[derive(Debug, Copy, Clone, FromPrimitive, Deserialize, Serialize)]
3 #[allow(non_camel_case_types, dead_code)]
4 pub enum Protocol {
5     J1850VPW = 0x01,
6     J1850PWM = 0x02,
7     ISO9141 = 0x03,
8     ISO14230 = 0x04,
9     CAN = 0x05,
10    ISO15765 = 0x06,
11    SCI_A_ENGINE = 0x07,
12    SCI_A_TRANS = 0x08,
13    SCI_B_ENGINE = 0x09,
14    SCI_B_TRANS = 0x0A,
15 }

```

and here is an example of how the Passthru function calls are converted from C++ to Rust:

```

1 #if defined __WIN32__
2     #define APICALL __stdcall
3 #else
4     #define APICALL
5 #endif
6 ...
7 typedef long APICALL (*J2534_PassThruWriteMsgs)(unsigned long ChannelID,
8                                                 PASSTHRU_MSG *pMsg, unsigned long *pNumMsgs, unsigned long Timeout);
9 ...

```

and here the function is declared in Rust:

```

1 #[no_mangle]
2 #[allow(non_snake_case)]
3 pub extern "stdcall" fn PassThruWriteMsgs(
4     ChannelID: u32,
5     pMsg: *const PASSTHRU_MSG ,
6     pNumMsgs: *mut u32 ,
7     Timeout: u32 ,
8 ) -> i32 {

```

The `#[no_mangle]` macro tells Rust's compiler to not mangle or modify the function name, but to keep it as is in order for the function to be exposed as part of a library and for it to be callable by external applications.

Windows ordinals

During testing, it was found that some applications such as Daimler's DAS software would call the Passthru API functions by their ordinal numbers rather than function name. An ordinal is a unique ID assigned to a function. Since Rust's compiler assigns a random ordinal number to each function by default, this would cause the custom Passthru library to not load, since the ordinal numbers did not match to the correct Passthru function. Therefore, a fix was to utilize a .def file with Windows' MSVC compiler in order to force Rust to assign the correct ordinal number to each function in the Passthru library:

```

1 LIBRARY
2 EXPORTS
3     PassThruOpen    @1
4     PassThruClose   @2
5     PassThruConnect @3
6     PassThruDisconnect @4
7     PassThruReadMsgs @5
8     PassThruWriteMsgs @6
9     PassThruStartPeriodicMsg @7
10    PassThruStopPeriodicMsg @8
11    PassThruStartMsgFilter @9
12    PassThruStopMsgFilter @10
13    PassThruSetProgrammingVoltage @11
14    PassThruReadVersion @12
15    PassThruGetLastError @13
16    PassThruIoctl @14

```

Then, using a custom build.rs file (Which gets executed by the compiler), its possible to tell the compiler to use the .def file during the compile process, but only under Windows:

```

1 use std::env;
2 fn main() {
3     let target_os = env::var("CARGO_CFG_TARGET_OS");
4     match target_os.as_ref().map(|x| &**x) {
5         Ok("macos") | Ok("linux") => {}
6         Ok("windows") => println!("cargo:rustc-cdylib-link-arg=/DEF:driver.def"),
7         tos => panic!("unknown target os {:?}", tos),
8     }
9 }

```

3.4.3 Communication between the adapter and driver

As mentioned in section 3.4, the Macchina M2 utilizes native USB in order to send and receive data, rather than an onboard UART to USB converter. The Serial functionality is instead provided by a virtual serial port driver. The Serialport-Rs library² provides an easy to use, cross platform Rust library for serial communication. Although there were initial issues with

²<https://gitlab.com/susurrus/serialport-rs>

the serial communication API under windows³, the library was able to function as expected and transfer data between the M2 and a PC, regardless of the operating system.

The following data structure was defined on both the firmware and Rust library, in order for both ends to send and receive data correctly. Below is the C++ definition in the adapter firmware:

```
1 #define COMM_MSG_SIZE 4096
2 #define COMM_MSG_ARG_SIZE COMM_MSG_SIZE-4
3
4 struct __attribute__((packed)) COMM_MSG {
5     uint8_t msg_id;
6     uint8_t msg_type;
7     uint16_t arg_size;
8     uint8_t args[COMM_MSG_ARG_SIZE];
9 };
```

The attributes in the message are used as follows:

1. **msg_id** - Unique identifier of the message. This ID will keep repeating in the range of 1-254, (0x01-0xFF), and is used by the PC and Firmware to identify a response message to a requested command. Each request and response message will have the same ID. Messages that do not require a response will have a message ID of 0x00.
 2. **msg_type** - A byte which identifies what kind of message is being sent by either the adapter or PC driver. See section C.2 for a full list of the message types and what they signify, as well as which end sends and receives them.
 3. **arg_size** - A 2 byte value indicating how long the parameters of the message are which reside in **args**
 4. **args** - The parameters or payload of the message itself.

Communication server

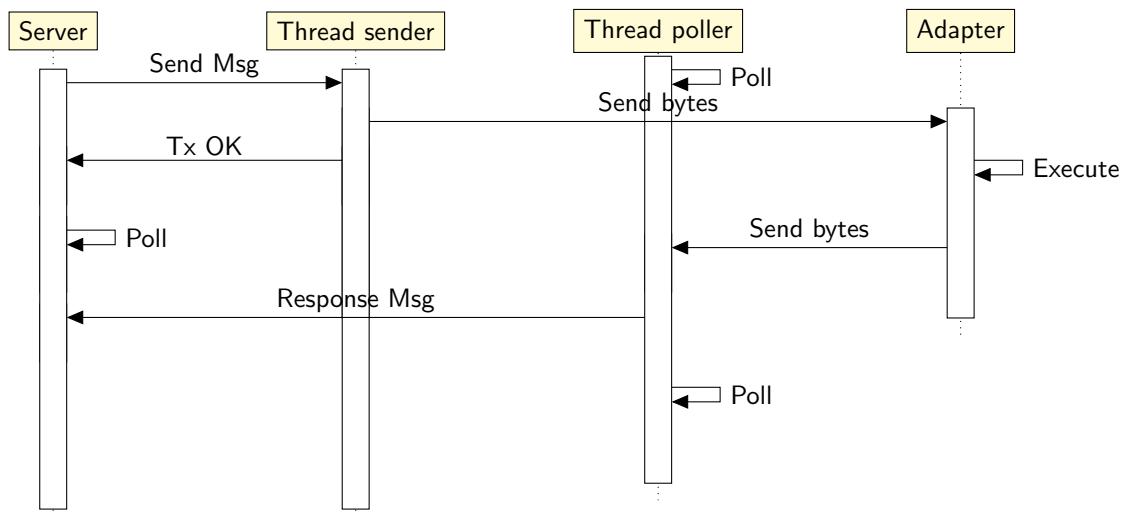


Figure 3.8: Expanded sequence diagram of communication server

³See B.1.1

Figure 3.8 shows an expanded view of the communication server sequence diagram. The 'server' endpoint is the interface other functions in the Passthru library can call to send messages to the Adapter. A sent message is sent to the server's sender thread, which then attempts to send the payload to the adapter. A maximum of 3 attempts are used to try and send the payload. If all 3 attempts fail, a Tx Fail status is sent back to the caller function, which then handles the appropriate action, as it assumes that the adapter is not connected to the PC. If payload sending was completed successfully, a Tx OK message is sent back to the caller function. In this event, the caller will now poll for a maximum of 2 seconds for a response message. Constantly in the background, there is the poller thread, which constantly polls the serial port for new bytes, reading them in order to ensure the serial port's buffer is not full (Linux has a maximum serial buffer size of 4096 bytes whereas Windows' is 16384 bytes). When a full payload is received by the poller thread, the response message is sent back to the sender via a thread channel. If the incoming message is a log message (See below), the poller instead logs the message to the libraries log file. In order to ensure the sender and receiver message are for the same operation, the server checks the message ID, which is unique to each IO operation which requires a response. If a response is not received in 2 seconds, the server terminates and returns a Timeout error back to the caller function. This Timeout is also sent back to the poller thread, which will then discard the received message if it does eventually get read. A timeout of 2 seconds is considered adequate as during testing, all commands usually get executed within 10ms.

3.4.4 Reading battery voltage

In order to read the battery voltage of the M2, there is a library called M2_12VIO⁴ that can be used to read the voltage on pin 16 of the OBD-II connector, which connects directly to the car's battery. However, this library is slightly inaccurate as the below table shows:

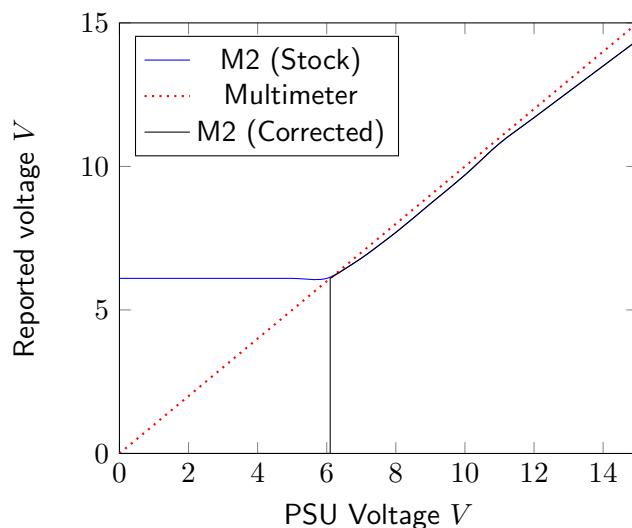


Figure 3.9: Voltage reading comparison between M2 (Stock and corrected) and Multimeter

As figure 3.9 shows, the reported voltage by the M2_12VIO library seems to level out at 6.1V, even if the actual supply voltage is lower than 6.1V. Therefore, the adapter firmware

⁴https://github.com/TDoust/M2_12VIO

(Black line) will cut off the voltage, interpreting any value less or equal to 6.1V as 0.0V. This way, when the adapter is unplugged from the vehicle, it will report 0V rather than 6.1V, which makes more sense.

3.4.5 ISO-TP Communication

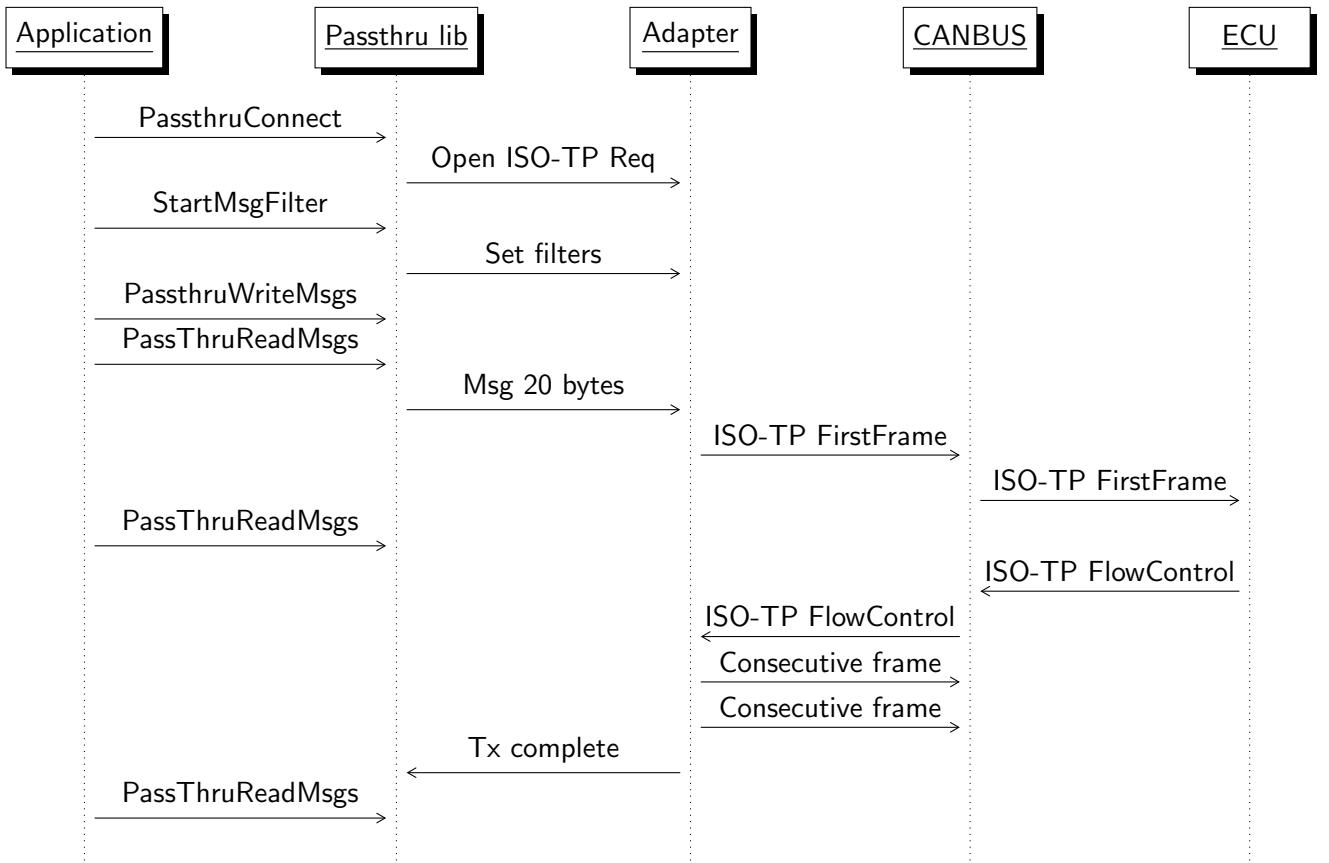


Figure 3.10: Sequence diagram for sending ISO-TP Data to an ECU

When transmitting ISO-TP data to an ECU. PassthruConnect is first called. This tells the adapter to open an ISO-TP Communication channel, with a specified baud rate, and also indications as to weather the CAN Network uses extended addressing or if ISO-TP uses extended addressing.

Then StartMsgFilter is called, this tells the adapter to apply 3 filters. Pattern and Mask (Which are both applied to listen to a specific CAN ID), and the Flow control filter. This filter is only used for ISO-TP and contains the CAN ID to transmit data on back to the ECU.

After this, PassthruWriteMsgs is called, where the application tries to transmit data to the ECU. This payload can be up to 4096 bytes long. The adapter receives this payload and sends the ISO-TP First frame to the ECU, initiating the data transfer.

The ECU Then responds back with a flow control frame. The adapter reads this, and applies the necessary block size and separation time, before transmitting ISO-TP consecutive frames to the ECU.

Once the transmission is complete, the adapter sends a message back to the Passthru library to indicate the data transfer is now complete. During the entire data transfer process, the Application has been polling the Passthru library for new data. When the data transfer

is complete, a blank message is returned to the Application, with the TX_MSG_TYPE bit set to 1, indicating data transfer complete.

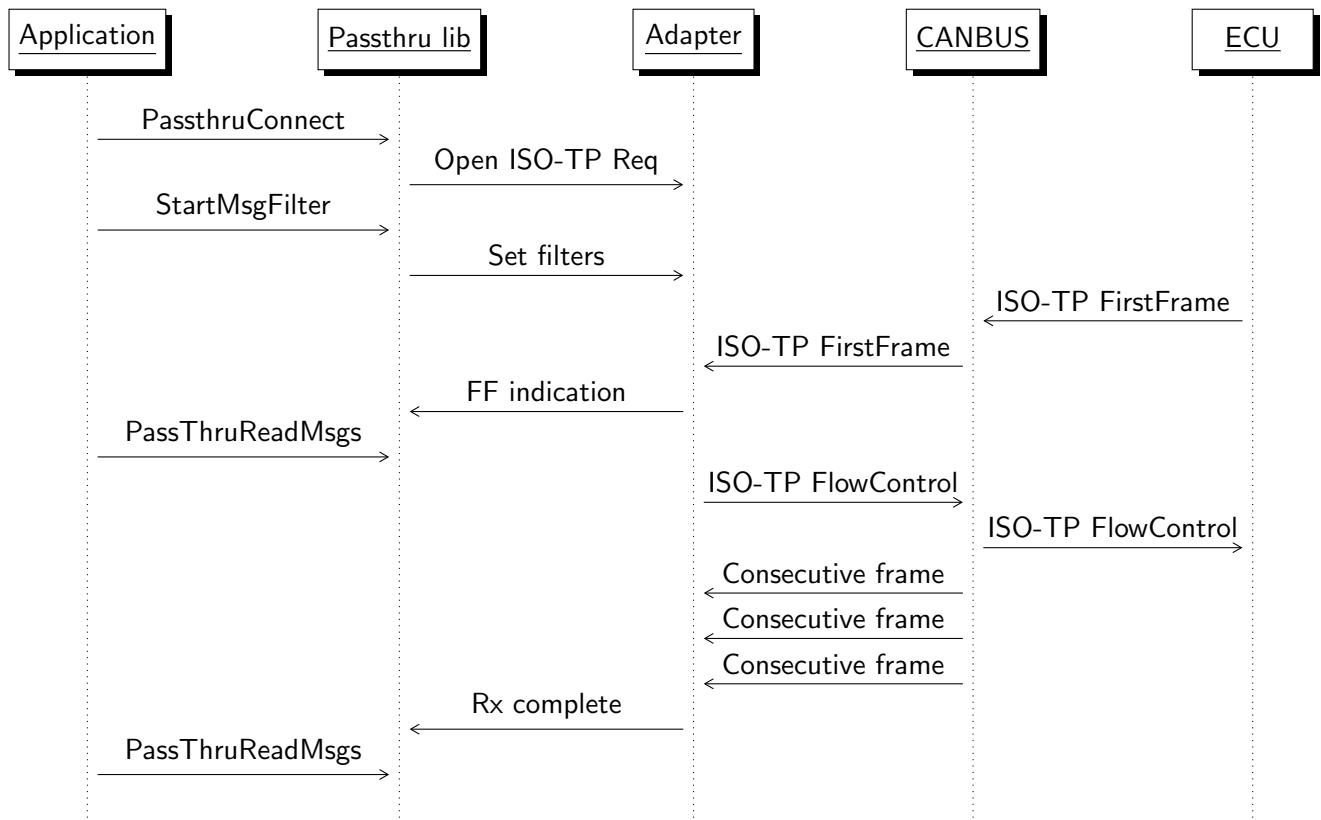


Figure 3.11: Sequence diagram for receiving ISO-TP Data from an ECU

When receiving data from an ECU, PassthruConnect and StartMsgFilter are called just like when transmitting data. Once the filters have been applied, the adapter can listen to incoming data from the ECU.

When adapter receives a first frame indication from an ECU, it sends a flow control frame back to the ECU, initiating the rest of the data transfer. At the same time, the adapter sends a message to the Passthru library which contains the CAN ID of the sending ECU with the ISO15765_FIRSTTF bit set. This message is then read by the user application and it will then begin polling for the full payload.

Upon receiving the flow control frame from the adapter, the ECU then sends all the consecutive CAN Frames which make up the full payload. Once the adapter verifies it has all the data, it sends the full payload back to the Passthru library, which is then read by the user application.

3.4.6 Porting the Passthru API to Linux and OSX

In order to port the J2534 API to Linux and OSX, 2 potential issues have to be overcome:

1. Mac OSX has a very tight security policy forbidding modification to its root filesystem. This means that a user cannot simply copy the library file to a directory like /usr/share/
2. Mac OSX and Linux have no equivalent to the Windows registry system

To work around the issue mentioned in 1, it was decided that the J2434 library driver and info data should reside in `~/.passthru/`. This is because on both Linux and OSX, writing to the users home directory (`~/`) does not require any elevated privileges.

To mitigate the section issue mentioned 2, it was decided the best approach would be to create a JSON file containing the attribute data defined in the J2534 API:

```

1 {
2     "CAN": true,
3     "ISO15765": true,
4     "ISO9141": true,
5     "ISO14230": true,
6     "SCI_A_TRANS": true,
7     "SCI_A_ENGINE": true,
8     "SCI_B_TRANS": false,
9     "SCI_B_ENGINE": false,
10    "J1850VPW": false,
11    "J1850PWM": false,
12    "FUNCTION_LIB": "~/.passthru/macchina.so",
13    "NAME": "Macchina M2 Under the dash",
14    "VENDOR": "rnd-ash@github.com",
15    "COM-PORT": "/dev/ttyACMO"
16 }
```

This means that every potential J2534 device would have its own JSON file stored in `~/.passthru/`. The above example being `~/.passthru/macchinam2.json` for the created Macchina M2 driver.

Every key name within this JSON is an exact copy of the key names found in the official J2534 specification for the Windows Registry.

From within the passthru library code itself, fetching the COM-PORT attribute now requires 2 code bases, one for Windows to locate the registry key within the Registry, and another for UNIX OS's to read the JSON file and find the COM-PORT attribute.

This was done with the conditional compilation flags on the library code to generate 2 versions of the `get_com_port` function, based on the compile target:

```

1 #[cfg(unix)]
2 fn get_comm_port() -> Option<String> {
3     if let Ok(content) = std::fs::read_to_string(shellexpand::tilde("~/.passthru/
4         macchina.json").to_string()) {
5         return match serde_json::from_str::<serde_json::Value>(content.as_str()) {
6             Ok(v) => v["COM-PORT"].as_str().map(String::from),
7             Err(_) => None
8         }
9     }
10    None
11 }
12 #[cfg(windows)]
13 fn get_comm_port() -> Option<String> {
14     if let Ok(reg) = RegKey::predef(HKEY_LOCAL_MACHINE).open_subkey("SOFTWARE\\
15     WOW6432Node\\PassThruSupport.04.04\\Macchina-Passthru") {
16         return match reg.get_value("COM-PORT") {
17             Ok(s) => Some(s),
18             Err(_) => None
19         }
20     }
21     None
22 }
```

Both functions work in the same way. They try to locate the COM-PORT key and read it. If the value was successfully read, then the COM-PORT value is returned, if it wasn't found, `None` is returned.

3.4.7 Logging activity

In order to aid debugging, log messages generated by either the M2 itself or passthru library are saved to a log file. On Linux, this Log file exists in `~/.passthru/macchina_log.txt` and on Windows it exists in `c:\Program Files (x86)\macchina\passthru\macchina_log.txt`.

Each log message generated has a unique tag:

Tag	Description
[DEBUG]	Messages for debugging only. These messages do not appear in release builds of the driver.
[ERROR]	Error messages, something has gone terribly wrong and the Passthru adapter cannot continue functioning.
[WARN]	An error has occurred somewhere, but the adapter can continue to function.
[INFO]	Info messages that can track what the adapter is doing
[M2LOG]	Messages that originate from the adapter itself

The generated log file has a dual purpose. It aids debugging of the library when used by an external application, but also shows in detail what the adapter is doing with regards to setting up CAN and ISO-TP channels, which can aid reverse engineering by snooping what a diagnostic application is requesting an adapter to listen for. An example of some log messages:

```

1 [INFO ] — PassthruOpen called
2 [DEBUG] — M2 serial writer thread starting!
3 [DEBUG] — M2 channel sender thread starting!
4 [DEBUG] — M2 serial reader thread starting!
5 [DEBUG] — Requesting channel open. ID: 0, Protocol: ISO15765, baud: 83333, flags: 0x0000
6 [M2LOG] — Standard CAN detected!
7 [M2LOG] — Normal ISO-TP Addressing detected!
8 [DEBUG] — Command took 15544us to execute
9 [DEBUG] — M2 opened channel!
10 [DEBUG] — Filter specified. Type: Mask filter , Data: [0, 0, 255, 255]
11 [DEBUG] — Filter specified. Type: Pattern filter , Data: [0, 0, 4, 244]
12 [DEBUG] — Filter specified. Type: Flow control filter , Data: [0, 0, 5, 180]
13 [DEBUG] — Setting ISO-TP flow control filter (ID: 0) on channel 0. Mask: [00, 00, FF, FF], Pattern: [00, 00,
    04, F4], FlowControl: [00, 00, 05, B4]

```

3.4.8 Performance optimizations with CAN Interrupts

During testing, it was found that the adapter would miss incoming CAN Frames on high traffic CAN Networks, with prevents ISO-TP receiving from working properly. This was because the default behavior of the M2's CAN Library is to have an Rx queue of CAN Frames, and pile all incoming CAN Frames onto the queue, which only has 30 slots. Once this queue is full, Any more incoming CAN Frames are simply dropped.

The solution to this problem was to register a callback for each of the 7 CAN mailbox's on the M2. When a frame is received by the mailbox, it triggers a hardware interrupt, which then executes the callback function. The callback function then places each incoming CAN Frame onto an assigned FIFO queue for each Mailbox.

Since the Passthru API can have 0-7 CAN Filters active at once (0-7 mailboxes with unique mask and pattern filters), the FIFO Queues are created dynamically in order to save memory. If a mailbox is not active, then no FIFO queue exists for it. Each FIFO queue is actually a Ring buffer, with a fixed size. Below is a few code snippets in order to allow for this unique interrupt handling:

```

1 // Each mailbox has a rxMailbox of 10 frames
2 #define MAX_RX_QUEUE 10
3 struct rxQueue {
4     volatile CAN_FRAME buffer[MAX_RX_QUEUE];
5     uint8_t head;
6     uint8_t tail;

```

```
7 };
```

Listing 3.1: RxQueue structure

```
1 void CustomCan::enableCanFilter(int id, uint32_t pattern, uint32_t mask, bool isExtended) {
2     if (id < 0 || id >= 7) return; // Invalid mailbox ID
3
4     // Set pattern and mask on the specified mailbox
5     Can0.setRXFilter(id, pattern, mask, isExtended);
6     // Delete any old buffer if it for some reason exists
7     __delete_check_rx_ring(id);
8     // Create our new ring
9     __create_check_rx_ring(id);
10    // Now register the callback so that frames get pushed to our mailbox
11 }
```

Listing 3.2: Function to enable CAN Filter. This sets up the RxQueue for the specified mailbox

```
1 void CustomCan::disableCanFilter(int id) {
2     if (id < 0 || id >= 7) return; // Invalid mailbox ID
3     Can0.setRXFilter(id, 0xFFFF, 0x0000, false);
4     __delete_check_rx_ring(id);
5 }
```

Listing 3.3: Function that removes the CAN Filter from the mailbox and deletes the RxQueue

```
1 void CustomCan::__rx_queue_push_frame(rxQueue &r, CAN_FRAME &f) {
2     uint8_t nextEntry = (r.head + 1) % MAX_RX_QUEUE;
3     // Queue is full, data is lost
4     if (nextEntry == r.tail) return;
5     memcpy((void *)&r.buffer[r.head], (void *)&f, sizeof(CAN_FRAME));
6     r.head = nextEntry;
7 }
```

Listing 3.4: Function that pushes a CAN Frame to a RxQueue during an interrupt

```
1 // This function gets called by the channels
2 bool CustomCan::receiveFrame(int mailbox_id, CAN_FRAME *f) {
3     if (mailbox_id < 0 || mailbox_id >= 7) return false; // Invalid mailbox ID
4     return __rx_queue_pop_frame(rxQueues[mailbox_id], *f);
5 }
6
7 bool CustomCan::__rx_queue_pop_frame(rxQueue &r, CAN_FRAME &f) {
8     // No frames in ring buffer
9     if (r.head == r.tail) return false;
10    memcpy((void *)&f, (void *)&r.buffer[r.tail], sizeof(CAN_FRAME));
11    r.tail = (r.tail + 1) % MAX_RX_QUEUE;
12    return true;
13 }
```

Listing 3.5: Functions which handles pulling CAN Frames from a specific mailboxes' RxQueue when requested.

```
1 void CustomCan::__callback_mb0(CAN_FRAME *f) { __rx_queue_push_frame(rxQueues[0], *f); }
2 void CustomCan::__callback_mb1(CAN_FRAME *f) { __rx_queue_push_frame(rxQueues[1], *f); }
3 void CustomCan::__callback_mb2(CAN_FRAME *f) { __rx_queue_push_frame(rxQueues[2], *f); }
4 void CustomCan::__callback_mb3(CAN_FRAME *f) { __rx_queue_push_frame(rxQueues[3], *f); }
5 void CustomCan::__callback_mb4(CAN_FRAME *f) { __rx_queue_push_frame(rxQueues[4], *f); }
6 void CustomCan::__callback_mb5(CAN_FRAME *f) { __rx_queue_push_frame(rxQueues[5], *f); }
7 void CustomCan::__callback_mb6(CAN_FRAME *f) { __rx_queue_push_frame(rxQueues[6], *f); }
```

Listing 3.6: Callback functions for each mailbox

3.5 Diagnostic GUI

As described in (1.4.3), the diagnostic application must be able to do the following things:

1. Allow the user to scan for UDS/KWP2000 compatible ECUs in their vehicle that rely on the ISO-TP transport protocol as described by [Nils Weiss, Sebastian Renner, Jürgen Mottok, Václav Matoušek (n.d.)]
2. Allow the user to select an ECU from their vehicle, and for the application to establish a diagnostic session with the ECU

3. Allow a user to run basic KWP2000/UDS commands on an ECU whilst the ECU is in its diagnostic session
4. have a simple CAN analyzer page for showing CAN Traffic on the OBD-II port
5. Use the OVD JSON (3.3) to run advanced functions on an ECU that are usually only able to be executed by commercial diagnostic software, and to interpret DTC Error codes and freeze frame data.

Due to the complexity of the user interface and underlying architecture, only the major points will be discussed in this report. Refer to D for the full code of the user interface.

3.5.1 Diagnostic server architecture

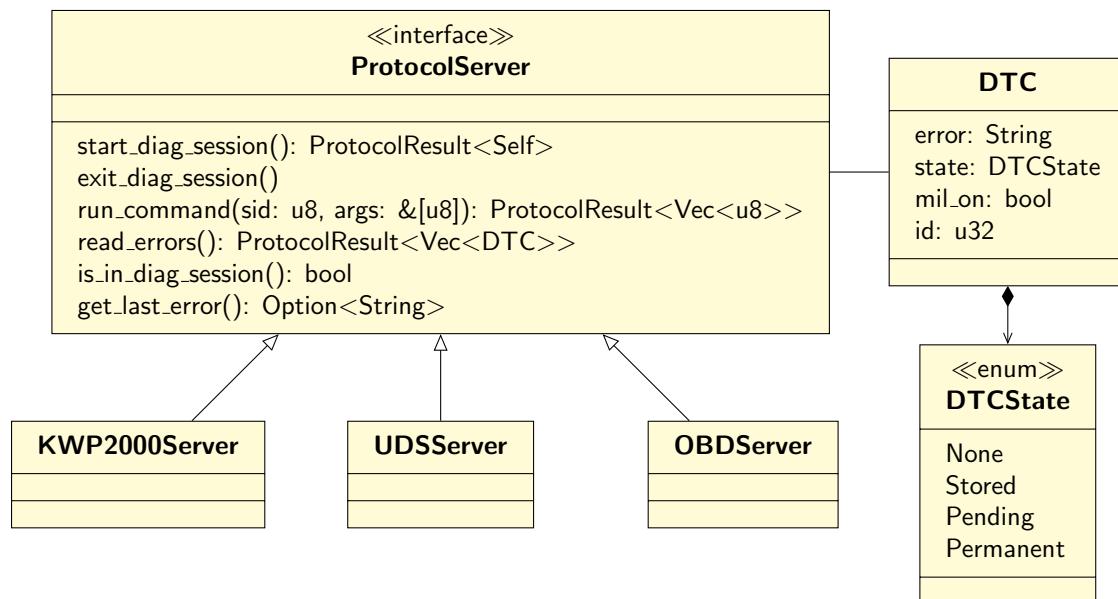


Figure 3.12: Diag servers UML overview

As discussed in (2.3), KWP2000, UDS and OBD all inherit the same request and response message format, with only the actual SID and PID's differing. Therefore, figure 3.12 shows how all three diagnostic servers inherit the same common interface. This will allow all diagnostic servers to work under a common UI interface, using dynamic dispatch to call the appropriate diagnostic server. Also to note. The storage format will be the same regardless of the protocol used. The DTC object can accommodate the DTC format of all 3 diagnostic protocols.

3.5.2 Communication server architecture

As discussed earlier, the idea of the communication server is to create a common server type system, which can use dynamic dispatch to load different adapter protocols. Although only SAE J2534 (Passthru) API implementation is planned for now, its worth creating a dynamic system so that in future, D-PDU and SocketCAN can be added easily to the application without major code modifications.

In its current form, the communication server UML is structured like so:

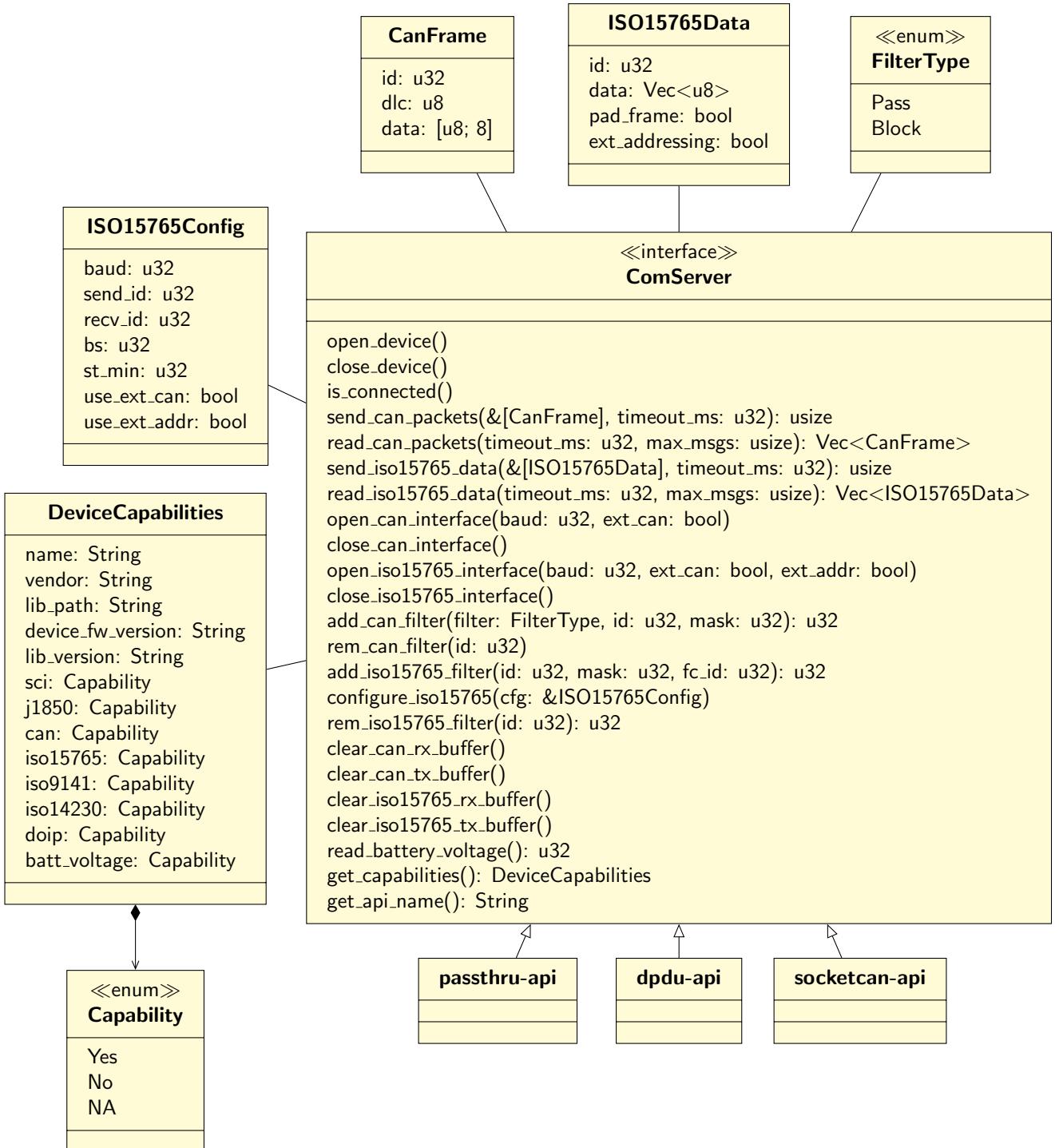


Figure 3.13: UML of ComServer

As seen in figure 3.13, there are multiple data classes to represent data such as CAN and ISO-TP payloads. This is done because each communication API has its own way of transmitting data to and from hardware with various communication protocols. Therefore, data types such as ISO15765Data exist as a portable data type that the rest of the application can work with, and the implementors of ComServer (Such as passthru_api) will internally convert to and from data types.

The **DeviceCapabilities** object is to be implemented by every API, and contains data about which functions the device supports. Currently, this does nothing, other than be displayed in the Applications home page. However, it is planned for future releases to use this capability data to automatically select the best protocol for communicating with an ECU. For an example, if the device does not support ISO14230, then there is no point in having a button in the user interface for the user to try and establish communication with the ECU using a protocol that is not supported by the adapter. The batt_voltage capability is currently used in order to enable or disable battery voltage monitoring. This avoids 0.0V being displayed in the user interface in the event the adapter does not support it. Instead, "Not supported" will be displayed in grey.

3.5.3 Implementation of the Passthru API

OpenVehicleDiags Passthru library implementation is an abstraction layer over the Passthru API. It is designed to use as little unsafe code as possible. The libloading⁵ library is used as a cross platform library loader, being able to load Windows DLLs and Linux SO files and OSX dynlib files. This library also contains functions to locate the library data automatically (Windows uses Registry, Linux/OSX uses JSON).

Passthru functions are declared as type aliases in the Passthru library like so:

```
1 pub type Result<T> = std::result::Result<T, j2534_rust::PassthruError>;
2 type PassThruCloseFn = unsafe extern "stdcall" fn(device_id: u32) -> i32;
```

The first line in this extract creates a type alias for the library. This way, if a function returns an error code (Rather than 0 - OK), it gets wrapped in Rust's return Err type. However if the function succeeded, then the Error's OK variant is returned, along with any data.

The second line shows an example of defining the Passthru API's functions as a type alias (Example shown is for PassThruClose).

The API also contains a struct which encapsulates all the library functions and reference to the loaded passthru library:

```
1 #[derive(Clone)]
2 pub struct PassthruDrv {
3     /// Loaded library to interface with the device
4     lib: Arc<libloading::Library>,
5     /// Open device connection
6     open_fn: PassThruOpenFn,
7     /// Close device connection
8     close_fn: PassThruCloseFn,
9     /// Connect a communication channel
10    connect_fn: PassThruConnectFn,
11    /// Disconnect a communication channel
12    disconnect_fn: PassThruDisconnectFn,
13    /// Read messages from a communication channel
14    read_msg_fn: PassThruReadMsgsFn,
15    /// Write messages to a communication channel
16    write_msg_fn: PassThruWriteMsgsFn,
17    /// Start a periodic message
18    start_periodic_fn: PassThruStartPeriodicMsgFn,
19    /// Stop a periodic message
20    stop_periodic_fn: PassThruStopPeriodicMsgFn,
21    /// Start a filter on a channel
22    start_filter_fn: PassThruStartMsgFilterFn,
23    /// Stop a filter on a channel
24    stop_filter_fn: PassThruStopMsgFilterFn,
25    /// Set programming voltage
26    set_prog_v_fn: PassThruSetProgrammingVoltageFn,
27    /// Get the last driver error description if ERR_FAILED
28    get_last_err_fn: PassThruGetLastErrorFn,
29    /// IOCTL
30    ioctl_fn: PassThruIoctlFn,
31    /// Get driver details
32    read_version_fn: PassThruReadVersionFn,
33 }
```

In order to call the Passthru API functions, the Passthru library contains wrapper functions such as the one shown below:

⁵https://github.com/nagisa/rust_libloading

```

1 //type PassThruCloseFn = unsafe extern "stdcall" fn(device_id: u32) -> i32;
2 pub fn close(&mut self, dev_id: u32) -> Result<()> {
3     let res = unsafe { (&self.close_fn)(dev_id) };
4     if res == 0x00 {
5         self.is_connected = false;
6     }
7     ret_res(res, ())
8 }
```

The function above shows the abstraction libraries implementation of PassthruClose. This function takes a mutable reference to the PassthruDrv struct (Similar to 'this' in other languages), as well as the device ID to close. The function returns a Result with OK variant type of unit. Then, the PassthruDrv's close function is called in an unsafe block. The unsafe block here is necessary as it is calling external code, which might not guarantee the same memory safety as Rust. If the function succeeded, the the OK Variant is returned. If the function call failed, then the Result's Err variant is returned, along with the error code.

Another example of the abstraction this library does is with PassthruReadMsgs function. This function is supposed to take a mutable reference to an array of PASSTHRU_MSGS, and attempts to fill the array up based on what the adapter can read. Instead, the abstraction library's implementation looks like the following:

```

1 //type PassThruReadMsgsFn = unsafe extern "stdcall" fn(channel_id: u32, msgs: *mut
2 //    PASSTHRU_MSG, num_msgs: *mut u32, timeout: u32) -> i32;
3 pub fn read_messages(&mut self, channel_id: u32, max_msgs: u32, timeout: u32) -> Result
4     <Vec<PASSTHRU_MSG>> {
5     let mut msg_count: u32 = max_msgs;
6     // Create a blank array of empty passthru messages according to the max we
7     should read
8     let mut write_array: Vec<PASSTHRU_MSG> = vec![
9         PASSTHRU_MSG::default();
10        max_msgs as usize
11    ];
12    // Call Passthru library PassthruReadMsgs
13    let res = unsafe {
14        (&self.read_msg_fn)(
15            channel_id,
16            write_array.as_mut_ptr() as *mut PASSTHRU_MSG,
17            &mut msg_count as *mut u32,
18            timeout,
19        )
20    };
21    // Error, but due to timeout waiting for more!
22    // Just return what we have and say OK!
23    if res == PassthruError::ERR_BUFFER_EMPTY as i32 && msg_count != 0 {
24        write_array.truncate(msg_count as usize);
25        return ret_res(0x00, write_array);
26    }
27    if msg_count != max_msgs {
28        // Trim the output vector to size
29        write_array.truncate(msg_count as usize);
30    }
31    ret_res(res, write_array)
32 }
```

This function simplifies the PassthruReadMsgs call significantly for the application. It internally constructs an array to pass a mutable pointer to PassthruReadMsgs itself, then return a Vector of PassthruMsgs that have been read by the adapter. If no messages are read, by default the Passthru library will return an error (ERR_BUFFER_EMPTY). This function can catch that error and simply return an empty array of messages, which is easier to handle in higher levels of the communication server.

3.5.4 User Interface

The Iced GUI library was utilized for the user interface, allowing for the user interface to be defined entirely in the rust codebase. Due to Iced being inspired by the Elm architecture, every page within the user interface has three main functions. An initialization function, an update function, which takes a Message type unique to each page, and a view function, which draws the user interface itself. Each page is stored in the form of a Struct with internal variables that can be only updated from the update function. To make the UI do something, the view function can place elements such as buttons, which emit a message which is sent to the update function. By default, Iced contains the following useful widgets:

- button
- Progress bar
- Drop-down
- Image view
- Radio button
- Checkbox
- Text view
- Text input view
- Scroll view
- Padding widget

Each element of the user interface can have its own custom 'style' struct applied to it, and also have its own alignment (Relative to its parent in the user interface)

Interface style

Iced has the ability to apply a 'style' struct on each element of the user interface. It was decided that as part of the user interface, there should be a dark theme. This will hopefully allow for less eye-strain when working with the application, especially in darker environments such as garages.

To theme the entire user interface, there is a global variable which indicates which theme is currently in use:

```

1 static mut CURR_THEME: Style = Style::Dark;
2 #[derive(Debug, Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
3 pub enum Style {
4     Light,
5     Dark,
6 }
7
8 pub fn set_dark_theme() {
9     unsafe { CURR_THEME = Style::Dark }
10 }
11
12 pub fn set_light_theme() {
13     unsafe { CURR_THEME = Style::Light }
14 }
15
16 pub fn toggle_theme() {
17     if *get_theme() == Style::Light {
18         set_dark_theme()
19     } else {
20         set_light_theme()
21     }
22 }
```

```

24 pub(crate) fn get_theme<'a>() -> &'a Style {
25     unsafe { &CURR_THEME }
26 }
```

For the main user interface components, builder functions were created which can apply, then return a newly created user interface element. Only helper for button creation will be shown, but a similar process occurs for other UI elements. The user interface is somewhat inspired by the Material design guidelines.

Buttons in the user interface comes in two flavours. Outlined, and Solid. Each button flavour has eight different themes. Primary colour, Secondary colour, Success, Danger, Warning, Info, Light and Dark.

To create the buttons, the following code is used:

```

1 pub enum ButtonType {
2     Primary,
3     Secondary,
4     Success,
5     Danger,
6     Warning,
7     Info,
8     Light,
9     Dark,
10 }
11
12 impl ButtonType {
13     pub(crate) fn get_colour(&self) -> Color {
14         match &self {
15             ButtonType::Primary => Color::from_rgb8(0x0d, 0x6e, 0xfd),
16             ButtonType::Secondary => Color::from_rgb8(0x66, 0x10, 0xf2),
17             ButtonType::Success => Color::from_rgb8(0x00, 0xb7, 0x4a),
18             ButtonType::Danger => Color::from_rgb8(0xf9, 0x31, 0x54),
19             ButtonType::Warning => Color::from_rgb8(0xff, 0xa9, 0x00),
20             ButtonType::Info => Color::from_rgb8(0x39, 0xc0, 0xed),
21             ButtonType::Light => Color::from_rgb8(0xfb, 0xfb, 0xfb),
22             ButtonType::Dark => Color::from_rgb8(0x26, 0x26, 0x26),
23         }
24     }
25 }
26
27 pub fn button_coloured<'a, T: Clone>(
28     state: &'a mut button::State,
29     text: &str,
30     btn_type: ButtonType,
31 ) -> Button<'a, T> {
32     let color = btn_type.get_colour();
33     Button::new(state, Text::new(text))
34         .style(ButtonStyle::new(color, false))
35         .padding(8)
36 }
```

The **ButtonType** enum has an implementation for each element to retrieve the desired colour of the button. These colours come from the Material design guidelines. When creating a button (`button_coloured`), a standard Iced button is created, then a style is applied to it, along with a desired padding.

The process of creating the style for the button looks like this:

```

1 pub struct ButtonStyle {
2     color: Color,
3     is_outlined: bool,
4 }
5
6 impl ButtonStyle {
7     pub fn new(color: Color, is_outlined: bool) -> Self {
8         Self { color, is_outlined }
9     }
10 }
11
12 impl button::StyleSheet for ButtonStyle {
13     fn active(&self) -> Style {
14         match super::get_theme() {
15             super::Style::Light => button::Style {
16                 shadow_offset: Default::default(),
17                 background: if self.is_outlined {
18                     WHITE.into()
19                 } else {
20                     self.color.into()
21                 },
22                 border_radius: BUTTON.RADIUS,
23                 border_width: if self.is_outlined {
24                     BUTTON_BORDER_WIDTH
25                 } else {
26                     0.0
27                 },
28             }
29         }
30     }
31 }
```

```

28     border_color: if self.is_outlined { self.color } else { WHITE },
29     text_color: if self.is_outlined { self.color } else { WHITE },
30   },
31   super::Style::Dark => button::Style {
32     shadow_offset: Default::default(),
33     background: if self.is_outlined {
34       DARK_BG.into()
35     } else {
36       self.color.into()
37     },
38     border_radius: BUTTON_RADIUS,
39     border_width: if self.is_outlined {
40       BUTTON_BORDER_WIDTH
41     } else {
42       0.0
43     },
44     border_color: if self.is_outlined { self.color } else { WHITE },
45     text_color: if self.is_outlined { self.color } else { WHITE },
46   },
47 }
48 }
49
50 fn hovered(&self) -> Style { /* Code not shown */ }
51 fn pressed(&self) -> Style { /* Code not shown */ }
52 fn disabled(&self) -> Style { /* Code not shown */ }

```

In this code block, the theme style for the button changes based on the global theme of the rest of the interface (Dark/Light). Then the returned button style is also modified based on if the button is supposed to be coloured (solid), or outlined. Not shown here is the styling for a hovered button, pressed button, or disabled button. All 3 follow the exact same method of creating the return style as the active button style, except with different colours.

Launcher

The launcher is responsible for enumerating located diagnostic adapters on the users computer, and to then display them to the user, allowing them to select which adapter to use with OpenVehicleDiag.

When the user selects Launch after selecting an adapter. A ComServer is created with the specified adapter API. If an error occurs during this process, then the launcher bails out and displays the error to the end user.



Figure 3.14: OpenVehicleDiag's launcher (Passthru device enumeration)

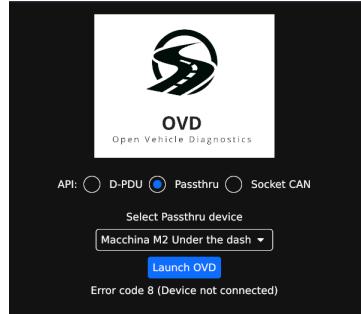


Figure 3.15: OpenVehicleDiag's launcher displaying Passthru error

As seen in figure 3.15, the Passthru library for the M2 was loaded, but returned an error as the M2 was not connected to the PC. This error is then displayed in the launcher.

The code to achieve this looks like the following in the Launcher's update function:

```

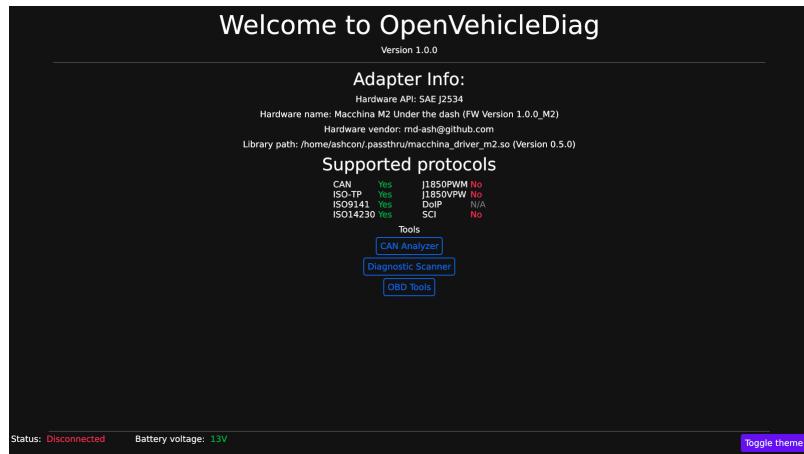
1 pub fn update(&mut self, msg: &LauncherMessage) -> Option<WindowMessage> {
2     match msg {
3         LauncherMessage::SwitchAPI(api) => self.api_selection = *api,
4         LauncherMessage::DeviceSelected(d) => {
5             if self.api_selection == API::Passthru {
6                 self.selected_device_passthru = d.clone()
7             } else if self.api_selection == API::DPdu {
8                 self.selected_device_dpdu = d.clone()
9             }
10        }
11        LauncherMessage::LaunchRequested => {
12            if self.api_selection == API::Passthru {
13                match self.get_device_passthru() {
14                    Ok((details, driver)) => {
15                        let mut server = PassthruApi::new(details, driver);
16                        if let Err(e) = server.open_device() {
17                            self.status_text = e.to_string()
18                        } else {
19                            // Ready to launch OVD!
20                            return Some(WindowMessage::StartApp(server.clone_box()));
21                        }
22                    }
23                    Err(x) => self.status_text = x.to_string(),
24                }
25            } else if self.api_selection == API::DPdu {
26                // TODO D-PDU Launching
27            }
28        }
29    }
30    None
31 }
```

This code block receives a signal 'msg' when a UI element is interacted with. In the case of "LaunchRequested", which occurs when the Launch button is pressed. When this occurs, if the API selection is Passthru, and the listed device was located, the PassthruAPI library attempts to load the device. If it was OK, then a message is emitted to the Main Window, telling it to launch the Home page. As part of this message, the dynamic ComServer (from the PassthruAPI) is passed to the main window.

If the loading of the Passthru device was unsuccessful, then the status_text is set to the error message returned by the Passthru API.

Home page

The home page is a gateway to all of OpenVehicleDiag's various functions and pages. It also displays details about the application, and adapter specifications being utilized.



The 'Supported Protocols' section is built by querying the ComServer's DeviceCapabilities value to see which communication interfaces are supported. 'Yes' (Green) indicates a method is supported by both the adapter API and the adapter itself. 'No' (Red) indicates that the API (EG: Passthru) does support the communication method, but the adapter itself does not. 'N/A' (Grey) indicates that the adapter API itself does not support the communication protocol.

Status bar

The status bar is a small part of the overall user interface. Taking its inspiration from Daimler's DAS software, it provides status on the connection to the vehicle (If an interface is currently open and communicating with a vehicle), and also the battery voltage of the vehicle. The status bar also provides a 'Go Home' button, and also a 'Back' button which allows users to quickly go back, or go to the home page. Both the 'Back' and 'Go Home' button can be disabled by any page in the user interface. This is used so that certain operations can be cancelled safely within the main user interface, rather than by clicking Go back or home, which forcibly destroys the current page.

The battery voltage is queried every two seconds. This is done by submitting 'events' to the main window every two seconds:

```

1 fn subscription(&self) -> Subscription<Self::Message> {
2 let mut batch: Vec<Subscription<WindowMessage>> = vec![];
3     if self.poll_voltage {
4         batch.push(
5             time::every(std::time::Duration::from_secs(2)).map(WindowMessage::StatusUpdate),
6         );
7     }
8     Subscription::batch(batch)
}

```

This allows for the battery voltage to be polled every two seconds, but to also be done on the same thread as the user interface, rather than a background thread. This is done because certain API's such as Passthru do not support multi-threading.

the variable 'poll_voltage' is set when the main window loads, by querying the ComServer's DeviceCapabilities. If battery voltage reading is supported, 'poll_voltage' is true, else it is false.

This is then utilized by the Status bar draw code:

```

1 let v = if self.poll_voltage {
2     if self.voltage < 12.0 && self.voltage > 11.5 {
3         text(format!("{}V", self.voltage).as_str(), TextType::Warning) // Amber
4     } else if self.voltage < 11.5 {
5         text(format!("{}V", self.voltage).as_str(), TextType::Danger) // Red
6     } else {
7         text(format!("{}V", self.voltage).as_str(), TextType::Success) // Green
8     }
9 } else {

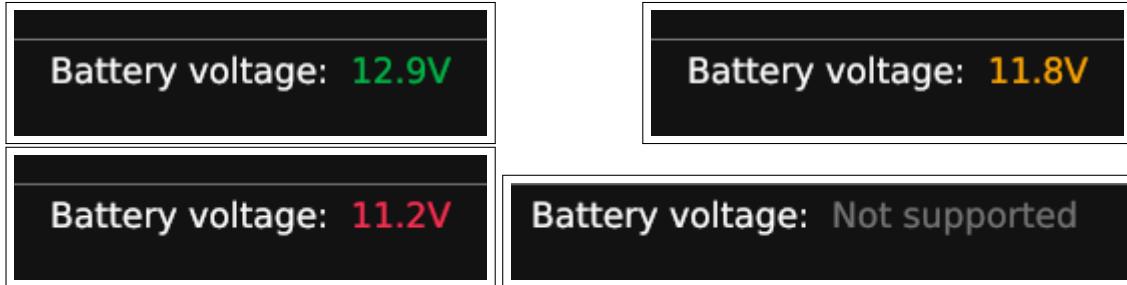
```

```

10     text("Not supported", TextType::Disabled) // Grey
11 };

```

As seen above, there are 4 styles for battery voltage display. If the voltage is between 11.5V and 12.0V, then the battery voltage colour will be Amber. If the voltage is less than 11.5V, the text is Red. Any voltage higher than 12.0V is displayed in Green. If battery voltage reading is not supported by the adapter, Grey text saying "Not Supported" will be displayed.



CAN Analyser

The CAN Analyser is a simple page which is designed to show CAN traffic on the OBD-II port. The user interface initially has a simple 'Connect' button. When pressed, the CAN Analyser will begin by setting up an open CAN interface on the ComServer, and wake up the OBD-II port by sending a OBD-II over CAN request message. This message is used since the vast majority of cars will respond to it, thus wake up the OBD-II port allowing for traffic to be monitored. This button is also togglable to disable the CAN Interface. When pressed, the following code is executed:

```

1 if self.is_connected { // Already connected — We are disconnecting CAN
2     if let Err(e) = self.server.as_mut().close_can_interface() {
3         self.status_text = format!("Error closing CAN Interface {}", e)
4     } else {
5         self.is_connected = false;
6         self.can_queue.clear();
7     }
8 }
9 }
10 // Try to open CAN Interface (Detect error if any)
11 else if let Err(e) = self.server.as_mut().open_can_interface(500_000, false) {
12     self.status_text = format!("Error opening CAN Interface {}", e)
13 }
14 // Opening CAN interface was OK!
15 else {
16     self.is_connected = true;
17     if let Err(e) =
18         self.server
19             .as_mut()
20                 .add_can_filter(FilterType::Pass, 0x0000, 0x0000)
21     {
22         self.status_text = format!("Error setting CAN Filter {}", e)
23         // Send OBD-II wakeup packet to wake up car's OBD-II port
24     } else if let Err(e) = self.server.send_can_packets(
25         &[CanFrame::new(
26             0x07DF,
27             &[0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00],
28         )],
29         0,
30     ) {
31         self.status_text = format!("Error sending wake-up packet {}", e)
32     }
33 }

```

Once connected, the CAN interface is polled every 10 milliseconds for new data:

```

1 pub fn subscription(&self) -> Subscription<TracerMessage> {
2     if self.is_connected {
3         return time::every(std::time::Duration::from_millis(10)).map(TracerMessage::NewData);
4     }
5     Subscription::none()
6 }

```

When the message is received, the following code is executed to push each CAN Frame to its own location within a HashMap

```

1 if let Ok(m) = self.server.as_ref().read_can_packets(0, 100) {
2     for f in m {
3         self.can_queue.insert(f.id, f);
4     }
5 }
```

Each hashMap entry uses the CAN ID as a unique key (Which will always be unique), with the CAN Data array being the value stored in the hashmap.

In order to display CAN Frames to the user, there are two available formats, which can be switched by using a checkbox:

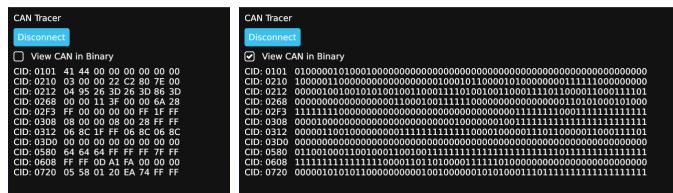


Figure 3.16: Standard CAN display (Hex) vs Binary CAN

The code to complete this looks like the following:

```

1 let mut col = Column::new();
2 let mut x: Vec = curr_data.keys().into_iter().copied().collect();
3 x.sort_by(|a, b| a.partial_cmp(b).unwrap());
4 for cid in x { // For each CAN ID
5     let mut container = Row::new();
6     container = container.push(
7         Row::new()
8             .push(Text::new(format!("CID: {:04X}", i.id)))
9             .width(Length::Units(100)),
10    );
11    for byte in i.get_data() { // Add bytes to view
12        container = match self.binary {
13            // Binary display
14            true => container.push(Row::new().push(Text::new(format!("{:08b}", byte)))),
15            // Hex display
16            false => container.push(
17                Row::new().push(Text::new(format!("{:02X}", byte))).width(Length::Units(30))),
18            .
19        };
20    }
21 }
```

3.5.5 KWP2000 and UDS implementations

As mentioned in section 2.3, KWP2000 and UDS are very similar. Therefore, they will both be covered in this section.

Each Diagnostic server uses the same function for sending and receiving command responses from an ECU over an ISO-TP connection:

```

1 fn run_command_iso_tp(
2     server: &dyn ComServer,
3     send_id: u32,
4     cmd: u8,
5     args: &[u8],
6     receive_require: bool,
7     response_timeout: u128,
8 ) -> std::result::Result<Vec<u8>, ProtocolError> {
9     let mut data = ISO15765Data {
10         id: send_id,
11         data: vec![cmd],
12         pad_frame: false,
13         ext_addressing: true,
14     };
15     data.data.extend_from_slice(args);
16     if !receive_require {
17         server
18             .send_iso15765_data(&[data], 0)
19             .map(|_| vec![])
20             .map_err(ProtocolError::CommError)
21     } else {
22         // Await max 1 second for response
23         let res = server.send_receive_iso15765(data, 1000, 1)?;
24         if res.is_empty() {
```

```

25         return Err(ProtocolError::Timeout);
26     }
27     let mut tmp_res = res[0].data.clone();
28     if tmp_res[0] == 0x7F && tmp_res[2] == 0x78 {
29         // ResponsePending
30         println!("DIAG — ECU is processing request — Waiting!");
31         let start = Instant::now();
32         while start.elapsed().as_millis() <= 2000 {
33             // ECU accepted the request, but cannot respond at the moment!
34             if let Some(msg) = server.read_iso15765_packets(0, 1)? .get(0) {
35                 tmp_res = msg.data.clone();
36             }
37         }
38         if tmp_res[0] == 0x7F {
39             // Still error :(
40             Err(ProtocolError::ProtocolError(Box::new(
41                 Self::Error::from_byte(tmp_res[2]),
42             )));
43         } else if tmp_res[0] == (cmd + 0x40) {
44             Ok(tmp_res)
45         } else {
46             eprintln!(
47                 "DIAG — Command response did not match request? Send: {:02X} — Recv: {:02X}",
48                 cmd, tmp_res[0]
49             );
50             Err(ProtocolError::Timeout)
51         }
52     }
53 }
54 }
```

This function starts by building an ISO15765Data payload to transmit to the ECU. The 'cmd' byte is the SID of the protocol, and args contain any additional data. If 'receive_response' is set, the server does not require a response from the ECU, so it immediately bails upon sending the data. If response from the ECU is required, it begins waiting for up to 2 seconds for a response from the ECU.

If the ECU responds with an error, it is returned as a ProtocolError. However, if the error code is 0x78, then the server begins a holding sequence, blocking the thread for up to another 2 seconds until the ECU responds. This is because the error 0x78 implies the ECU has accepted the request, but is unable to respond to the request immediately. Both KWP2000 and UDS state that in this event, the diagnostic server is to stop sending any data to the ECU (Including TesterPresent commands), until the ECU responds, or a timeout occurs. This second timeout is 2 seconds. If the ECU responds with a positive response, the response is returned as a byte array back to whichever diagnostic server sent the message.

Utilizing both the KWP2000 specification [DaimlerChrysler (2002)] and UDS specification [ISO (2006)], basic diagnostic functions were implemented such as clear / Read DTCs, as well as some additional functions for KWP2000 which allow for querying detailed ECU software and hardware version data.

For reading DTCs from ECUs, each protocol returns a different data format, so the Diagnostic servers consolidate the data into a generic DTC structure. Below is the code to read DTCs with KWP2000:

```

1 fn read_errors(&self) -> ProtocolResult<Vec<DTC>> {
2     // 0x02 — Request Hex DTCs as 2 bytes
3     // 0xFF00 — Request all DTCs (Mandatory per KWP2000)
4     let mut bytes = self.run_command(Service::ReadDTCByStatus.into(), &[0x02, 0xFF, 0x00])?;
5     bytes.drain(..1);
6     let count = bytes[0] as usize;
7     bytes.drain(0..1);
8
9     let mut res: Vec<DTC> = Vec::new();
10    for _ in 0..count {
11        let name = format!("{:02X}{:02X}", bytes[0], bytes[1]);
12        let status = bytes[2];
13        //let flag = (status >> 4 & 0b00000001) > 0;
14        //0b011
15        let storage_state = (status >> 5) & 0b00000011;
16
17        let state = match storage_state {
18            1 => DTCState::Stored,
19            2 => DTCState::Pending,
20            3 => DTCState::Permanent,
21            _ => DTCState::None
22        };

```

```

23     // Is check engine light on?
24     let mil = (status >> 7 & 0b00000001) > 0;
25
26     res.push(DTC {
27       error: name,
28       state,
29       check_engine_on: mil,
30       id: ((bytes[0] as u32) << 8) | bytes[1] as u32
31     });
32     bytes.drain(0..3); // DTC is 3 bytes (1 for status, 2 for the ID)
33   }
34   Ok(res)
35 }
36 }
```

3.5.6 Automated ECU Scanner

The Automated ISO-TP ECU Scanner is the main part of OpenVehicleDiag. Based on the work done in [Nils Weiss, Sebastian Renner, Jürgen Mottok, Václav Matoušek (n.d.)], it shows it is possible to detect ISO-TP endpoints in a vehicle by sending bogus ISO-TP First frames to the ECU, as it will always respond with a flow control message, regardless if the incoming data is valid or not.

Prior to the scan, a warning message must be presented to the user. This message is to notify the users in advance that many warning lights might briefly illuminate on their instrument cluster, as will be shown later on.

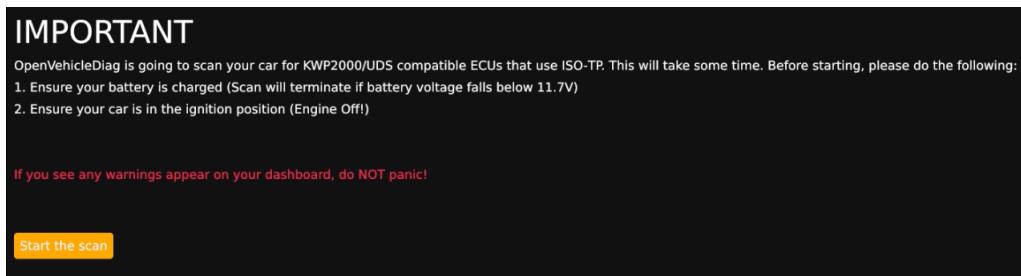


Figure 3.17: Warning message presented to the user prior to the ECU scan

To begin the scan, the scanner begins by setting up an open CAN Interface on the applications ComServer, and configures the filter to be open to all incoming traffic:

```

1 self.server.open_can_interface(500_000, false);
2 self.server.add_can_filter(commapi::comm_api::FilterType::Pass, 0x00000000, 0x00000000);
```

This allows for OpenVehicleDiag to receive ALL incoming CAN Packets from the OBD-II port as it is an open filter.

Next, an OBD-II request is sent to the OBD-II ports CAN interface. This is done to wake-up the CAN Interface on the vehicles side:

```
1 self.server.send_can_packets(&[CanFrame::new(0x07DF, &[0x09, 0x02])], 0)
```

The next step is important. In order to ensure that OpenVehicleDiag does not accidentally send a CAN Frame to a vehicle which already exists on the bus, it listens to traffic on the OBD-II port for 15 seconds. During this listening period, any incoming CAN Traffic is noted, and the CAN ID's of the incoming traffic are added to a blacklist of CAN ID's to not send to the vehicle for probing ISO-TP endpoints.

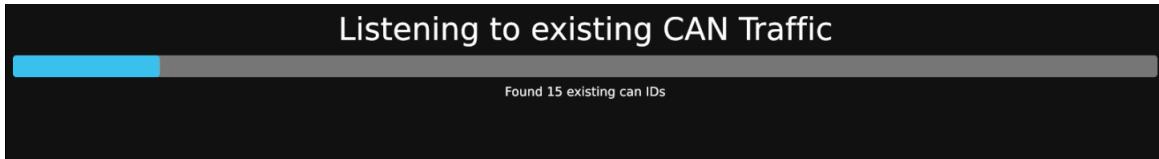


Figure 3.18: Listing to existing CAN traffic

After the initial poll period, the scan begins iterating over every CAN ID between 0x000 and 0x7FF (CAN's Maximum CAN ID with Standard 11 bit addressing). During every iteration, it will send a fake ISO-TP Start frame, and poll for 100ms for incoming data that is not in the existing blacklist. If the incoming data looks like an ISO-TP Flow control message, it is added to an array, along with the Send CAN ID used.

```

1 self.get_next_canid();
2 self.server.clear_can_rx_buffer();
3 // Send a fake ISO-TP first frame. Tell the potential ECU we are sending 16 bytes to it. If it uses ISO-TP, it
4 // will send back a
5 self.server.send_can_packets(
6     &[CanFrame::new(
7         self.curr_scan_id,
8         // Fake payload to target address. Sending 0x10 bytes of data (16 bytes)
9         &[0x10, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00],
10     )],
11     0,
12 );
13 self.clock = Instant::now();
14 while self.clock.elapsed().as_millis() < 100 {
15     // Keep polling CAN!
16     for frame in &self.server.read_can_packets(0, 10000).unwrap_or_default() {
17         if self.can_traffic_id_list.get(&frame.id).is_none() {
18             // Its a new frame we haven't seen before!
19             let payload = frame.get_data();
20             if payload[0] == 0x30 && payload.len() == 8 {
21                 // Possible recv ID? - It might pick up multiple IDs during the scan, we filter it later on
22                 if let Some(r) = self.stage2_results.get_mut(&self.curr_scan_id) {
23                     r.push(frame.id)
24                 } else {
25                     self.stage2_results.insert(self.curr_scan_id, vec![frame.id]);
26                 }
27             }
28         }
29     }
30 }
```

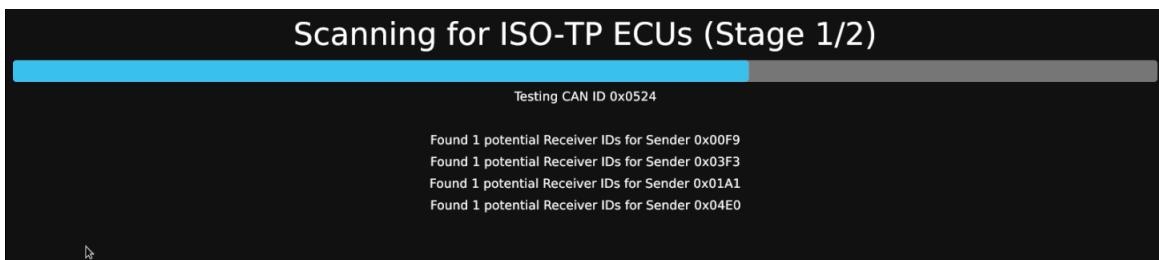


Figure 3.19: Locating potential ISO-TP endpoints

Once this initial list of potential ISO-TP endpoints has been created, it is then iterated over again, but this time using a specified CAN Filter in order to remove any false positives. If the incoming frame is seen again with the CAN filter applied, then it is definitely an ISO-TP endpoint.

```

1 let keys: Vec<u32> = self.stage2_results.keys().copied().collect();
2 let filter_id = self
3     .stage2_results
4     .get(&keys[self.curr_scan_id as usize])
5     .unwrap();
6 self.filter_idx = self
7     .server
```

```

8     // Specified filter with our specific CAN ID we are searching for.
9     .add_can_filter(commapi::comm_api::FilterType::Pass, filter_id[0], 0xFFFF)
10    .unwrap();
11  self.server.send_can_packets(
12    &[CanFrame::new(
13      keys[self.curr_scan_id as usize],
14      &[0x10, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00],
15    ), 0];
16  for frame in &self.server.read_can_packets(0, 10000).unwrap_or_default() {
17    let payload = frame.get_data();
18    if payload[0] == 0x30 && payload.len() == 8 {
19      // True positive! We can add the Config to list!
20      self.stage3.results.push(ISO15765Config {
21        baud: 500000,
22        // -1 is current scan ID whilst in this loop
23        send_id: *keys.get((self.curr_scan_id - 1) as usize).unwrap(),
24        recv_id: frame.id,
25        block_size: payload[1] as u32,
26        sep_time: payload[2] as u32,
27        use_ext_isotp: false,
28        use_ext_can: false
29      })
30    }
31  }

```

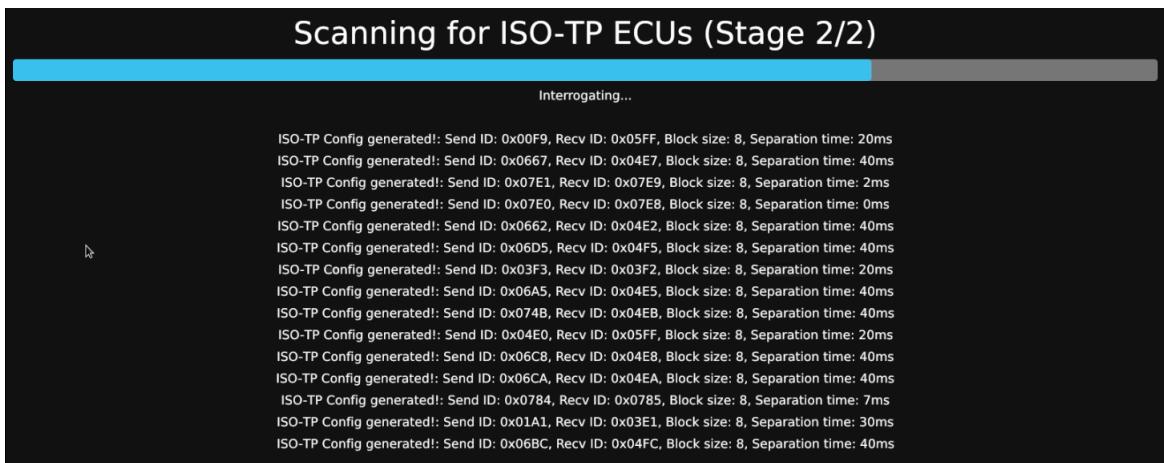


Figure 3.20: Finalizing ISO-TP scan results

At this point, a full list of positive ISO-TP endpoints on the vehicle have been recorded. Next, the scanner needs to determine which diagnostic protocol each endpoint supports. For each ISO15765 Config generated in the ISO-TP scan, it is iterated over, and both a KWP2000 and UDS diagnostic server are created for the ISO-TP endpoint. Due to the architecture of both servers, if the ECU rejects the start diagnostic session command ([0x10, 0x03] for UDS, [0x10, 0x92] for KWP2000), then the diagnostic server will return an Error upon initialization, and therefore the scanner knows the ECU does not support the protocol.

```

1 let ecu = self.stage3.results[self.curr_scan_id as usize];
2
3 let mut ecu_res = ECUDiagSettings {
4   name: "Unknown ECU name".into(),
5   send_id: ecu.send_id,
6   flow_control_id: ecu.recv_id,
7   block_size: ecu.block_size,
8   sep_time_ms: ecu.sep_time,
9   uds_support: false,
10  kwp_support: false,
11 };
12
13 // Interrogate the ECU with KWP2000 extended diagnostic session
14 match KWP2000ECU::start_diag_session(self.server.clone(), &ecu, None) {
15   Ok(mut s) => {
16     if let Ok(id) = read_dcx_mmc_id(&s) {
17       ecu_res.name = format!("ECU Part number: {}", id.part_number);
18       println!("ECU 0x{:04X} supports KWP2000!", ecu.send_id);
19       ecu_res.kwp_support = true;
20     }
21   s.exit_diag_session();
22 }

```

```

23     Err(e) => {
24         println!("KWP2000 server failed! {:?}", e);
25     }
26 }
27 // Interrogate the ECU with UDS extended diagnostic session
28 match UDSECU::start_diag_session(self.server.clone(), &ecu, None) {
29     Ok(mut s) => {
30         // TODO find a UDS only CMD to test with
31         println!("ECU 0x{:04X} supports UDS!", ecu.send_id);
32         self.stage4_results[self.curr_scan_id as usize].uds_support = true;
33         s.exit.diag_session();
34     }
35     Err(e) => {
36         println!("UDS server failed! {:?}", e);
37     }
38 }
39 self.curr_scan_id += 1;

```

Since KWP2000 has a standard way of reading the ECU part number, Line 16 shows that the KWP2000 scan attempt also tries to grab the part number from the ECU. If successful, this part number replaces the default "Unknown ECU Name" text. This is because this part number can be easily searched online to find out what ECU it is.

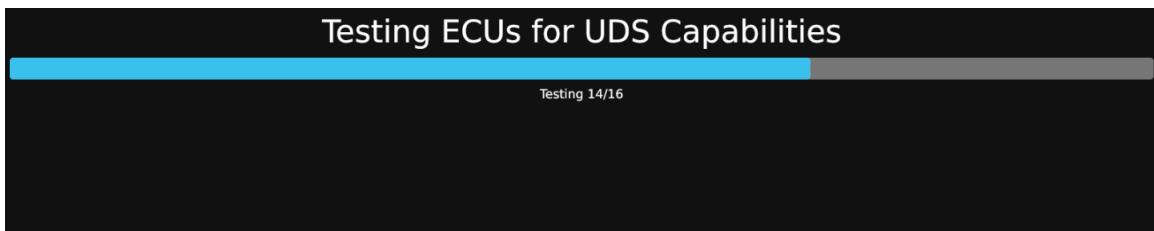


Figure 3.21: Scan progress for UDS compatible ECUs

As mentioned above, the reason for showing a large warning to the end user prior to starting the scan is because putting ECUs into diagnostic mode will cause them to partially deactivate. On modern cars, this can result in hundreds of warning lights illuminating on the users instrument cluster. These warning messages disappear after a couple seconds (As the ECU returns into its default session state).



Figure 3.22: Instrument cluster warning lights being displayed during the final stages of ECU detection

The last stage is to present the results of the scan to the user, and save the results to a JSON File.

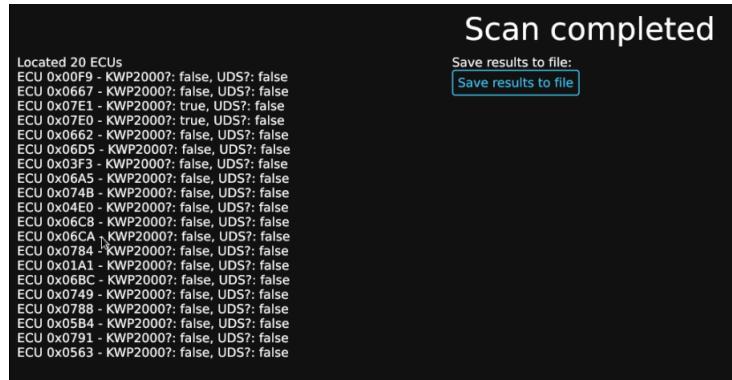
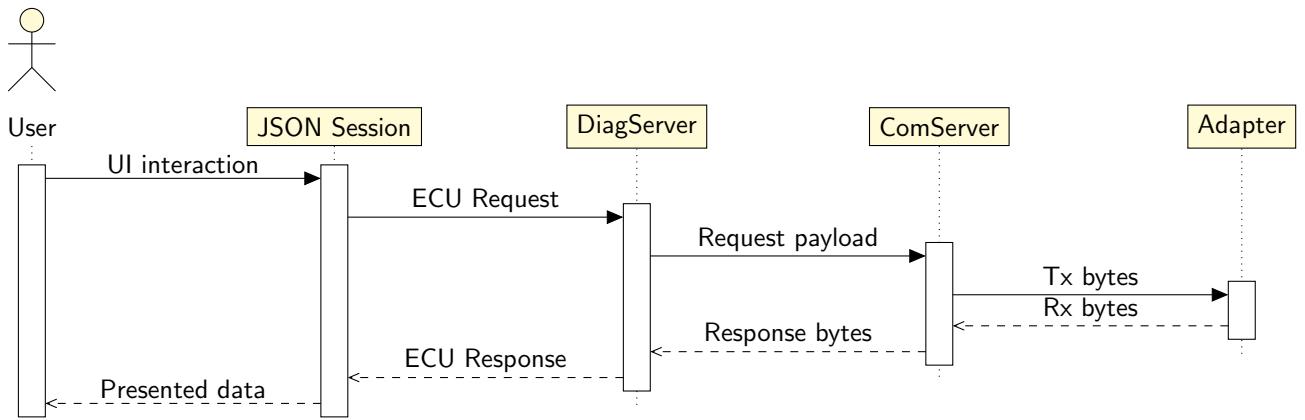


Figure 3.23: Results page of ECU Scanner

3.5.7 JSON Diagnostic session

The JSON diagnostic session UI is what allows the OpenVehicleDiag JSON to be used for diagnostics. It is the highest level of the diagnostic architecture for the application. As a sequence diagram, it communicated with the ECU like so:



When a JSON file is loaded into OpenVehicleDiag, it is automatically deserialized into the code structs described in section 3.3. From here, the diagnostic session server type and communication parameters are setup and applied. Next, OpenVehicleDiag attempts to read the variant ID from the ECU. This returns the software ID of the ECU. It is then matched against the variant list in the JSON. If nothing is found, the JSON session bails out and returns an error "No matching ECU variants discovered". If an ECU variant is located, the data about that variant is loaded into the JSON diagnostic session and the session begins.

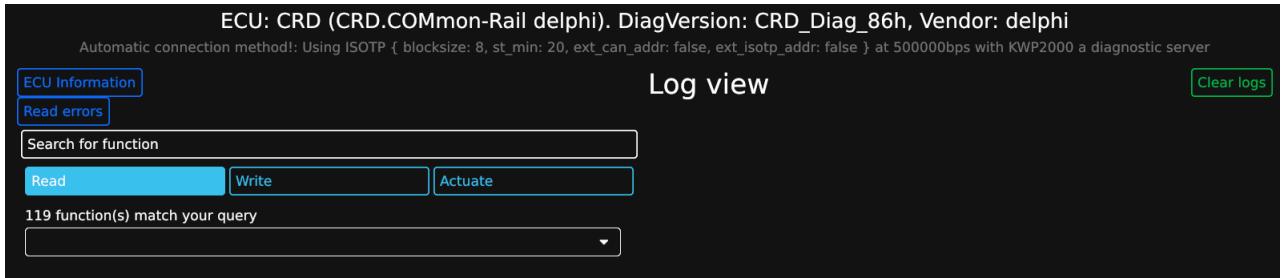


Figure 3.24: JSON Session home with CRD ECU

As the above image shows, this is the hom page of the JSON diagnostic session. At the top is the name of the ECU along with the vendor name and software version of the ECU. This is all found within the JSON.

```

1 let diag_server_type = match connection_settings.server_type {
2   common::schema::ServerType::UDS => DiagProtocol::UDS,
3   common::schema::ServerType::KWP2000 => DiagProtocol::KWP2000
4 };
5 println!("Detect. ECU uses {:?}", diag_server_type);
6
7 // For now, Diag server ONLY supports ISO-TP, not LIN!
8 let create_server = match connection_settings.connection_type {
9   ConType::ISOTP { blocksize, st_min, ext_isotp_addr, ext_can_addr } => {
10     let cfg = ISO15765Config {
11       baud: connection_settings.baud,
12       send_id: connection_settings.send_id,
13       recv_id: connection_settings.recv_id,
14       block_size: blocksize,
15       sep_time: st_min,
16       use_ext_can: ext_can_addr,
17       use_ext_isotp: ext_isotp_addr
18     };
19     // Dynamic diagnostic server creation
20     DiagServer::new(comm_server, &cfg, connection_settings.global_send_id diag_server_type)
21   },
22   ConType::LIN { ... } => return Err(SessionError::Other("K-Line is not implemented at this time".into()))
23 };

```

The above code reads the JSON data about connection info before building an ISOTP configuration setting, and applying it to a special 'DiagServer' class. This class is a dynamic wrapper around both the KWP2000 and UDS diagnostic servers.

DTC Errors

When the 'read errors' button is pressed, the JSON Session requests the underlying diagnostic server to read all the DTCs stored on the ECU. Then, for every DTC which is read, the session will then query freeze frame data about it. This returns both a DTC struct, and a sequence of bytes which represents the DTC Freeze frame request response message. Using the decoder function within the JSON, as mentioned earlier in section 3.3, the freeze frame bytes are fed through a decoder for the DTC's freeze frame array, in order to extract data about the freeze frame. Also, each read DTC's ID is cross-referenced with the DTC list in JSON. If a matching name is found, the error description is also presented to the end user. This is all shown in the DTC error view:

ECU Error view

Error	Description	State	MIL on
2054	Fuel temperature sensor-HP Pump-Electrical fault	Permanent	NO
2035	Engine coolant temperature sensor-Engine Bloc-Electrical fault	Permanent	NO
2086	Air pressure P1 sensor-In AMF-Electrical fault	Permanent	NO
23F4	VGT actuator-none-Driver fault	Permanent	NO
2379	Oil temperature sensor-Engine Block-Electrical fault	Permanent	NO
236A	EGR Cooler by pass actuator -none-electrical fault	Permanent	NO
206D	Inlet Air temperature sensor-In AMF-Electrical fault	Permanent	NO
22A5	Exhaust temperature-sensor before turbine-Electrical fault	Permanent	NO
20D3	EGR actuator-none-Driver fault	Permanent	NO
20E7	Inlet Metering Valve-HP Pump-Driver fault	Permanent	NO
201A	Boost pressure sensor-none-Electrical fault	Permanent	NO
22A0	High Pressure valve-Rail-Electrical fault	Permanent	NO
227C	SWIRL valve actuator-none-Driver fault	Permanent	NO

Freeze frame data

Engine Speed	0 rpm
U-Daten 1. Auftreten	0
Battery Voltage	11.921568 V
Boost Pressure Actuator duty cycle	49.80392 %
Boost Pressure	0.9803922 bar
Inlet Air Temperature	20 °C
Total Fuel Demand	0 mg/Hub
Engine state	SMC_STOPPED
Total Distance	0 Km

The user can click on each DTC, and the freeze frame table will display the interpreted freeze-frame data from the ECU. There is also a button to clear DTCs stored on the ECU. Since this utilizes KWP2000 or UDS, it is indeed possible to clear Permanent DTCs, unlike with generic OBD-II applications.

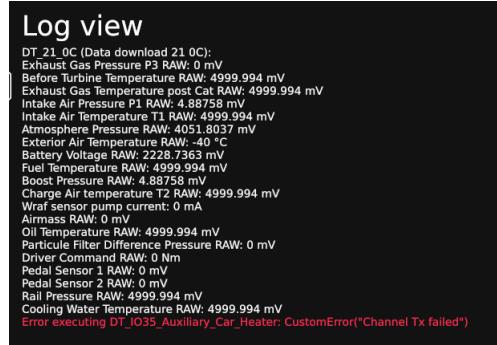
Reading ECU data

In OpenVehicleDiag's current state, only Read functions are executable (Reading data from an ECU using a defined payload without any additional user input). Loaded from JSON are a list of readable functions for the ECU:

Search for function

Read	Write	Actuate
119 function(s) match your query		
DT_I033_Exhaust_Gas_Recirculation DT_Result_Noisy_Test_Injector_4 DT_21_1A DT_I038_Wheel_Circumference DT_I053_Inlet_Metering_Valve DT_I06A_tlAds_Total_Time_In_State_Adsoption DT_I056_Glow_Indication_Lamp DT_30_23 DT_I030_Start_Torque DT_Result_Noisy_Test_Injector_1 DT_21_1E DT_21_0C DT_I028_Digital_Inputs_2_high_memory DT_I05E_Terminal_61_Generator_Lamp DT_I051_Oxygen_Sensor_Pre_Cat DT_30_C0 DT_I035_Auxiliary_Car_Heater DT_31_04		

When selected and executed, the log view displays the outputted presentation data using the decoder functions found in section 3.3. If the ECU responds negatively to the request, a human readable error message is displayed in red:



The screenshot shows a 'Log view' window with a black background and white text. At the top, it says 'Log view'. Below that is a list of sensor readings, each consisting of a name and its corresponding RAW value. The list includes: DT_21_0C (Data download 21_0C), Exhaust Gas Pressure P3 RAW: 0 mV, Before Turbine Temperature RAW: 4999.994 mV, Exhaust Gas Temperature post Cat RAW: 4999.994 mV, Intake Air Pressure P1 RAW: 4.88758 mV, Intake Air Temperature T1 RAW: 4999.994 mV, Atmosphere Pressure RAW: 4051.8037 mV, Exterior Air Temperature RAW: -40 °C, Battery Voltage RAW: 2228.7363 mV, Fuel Temperature RAW: 4999.994 mV, Boost Pressure RAW: 4.88758 mV, Intake Air Temperature T2 RAW: 4999.994 mV, Wraf sensor temp current: 0 mA, Airmass RAW: 0 mV, Oil Temperature RAW: 4999.994 mV, Particulate Filter Difference Pressure RAW: 0 mV, Driver Command RAW: 0 Nm, Pedal Sensor 1 RAW: 0 mV, Pedal Sensor 2 RAW: 0 mV, Rail Pressure RAW: 4999.994 mV, Cooling Water Temperature RAW: 4999.994 mV. At the bottom of the list, there is a red error message: 'Error executing DT_I035_Auxiliary_Car_Heater: CustomError("Channel Tx failed")'

These error messages come from either the ComServer (Indicating something wrong with the adapter), or DiagServer (Indicating the ECU Rejected the request message).

3.6 Summary

As shown in this chapter, it has been shown that it is possible to implement all 3 major parts of this project, and all whilst keeping the code modular enough to be easily expanded on at a later date.

Especially with OpenVehicleDiag's UI, there are a lot of details not covered by this report, mainly due to the length of the implementation, however this report has covered the major points which would be useful to 99% of end users when it comes to car diagnostics.

Chapter 4

Results, Discussion and Analysis

In this chapter, results and conclusions of utilizing OpenVehicleDiag with be discussed, as well as validating that the Open source J2534 (Passthru) driver works with other software that utilizes the Passthru protocol

4.1 Passthru driver

Due to time constraints, as as mentioned in section 3.4, the driver is currently limited to CAN and ISO-TP protocols, since there was not enough time to implement all 6 protocols that the J2534 specification uses. However, this adapter is still in its current form J2534 compliant, since the driver details indicate the adapter currently only supports CAN and ISO-TP.

For verifying J2534 compliance, a copy of Mercedes' DAS diagnostic system was utilized (**D**iagnostic **A**sistance **S**ystem), which supports the J2534 API.

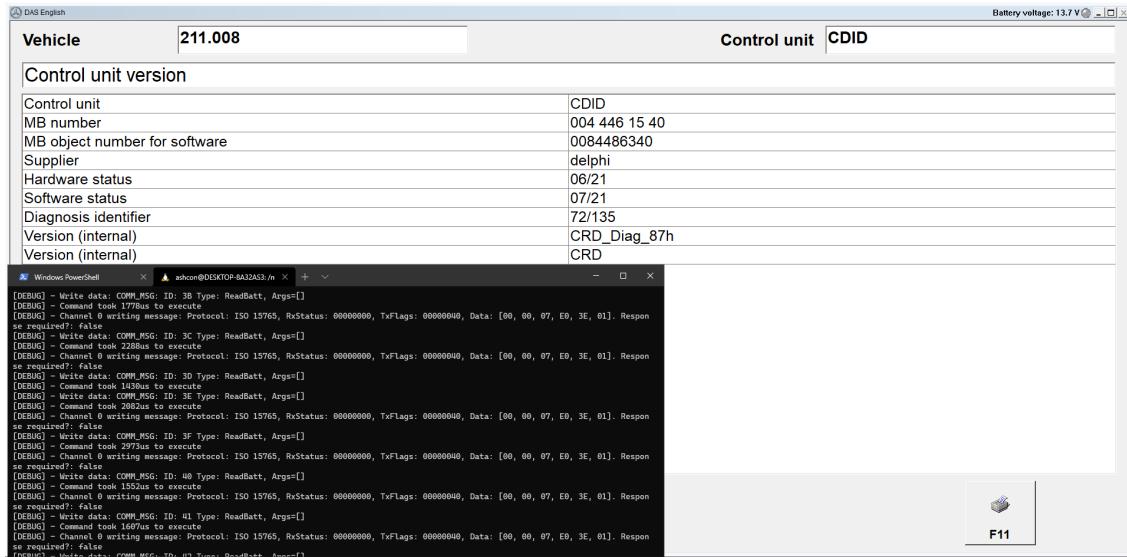


Figure 4.1: DAS utilizing the custom J2534 adapter

However, DAS only transmit small ISO-TP payloads to and from the ECU. Nothing comes close to the maximum 4096 byte limit of the ISO-TP protocol. Therefore, in order to verify that the driver and adapter can indeed handle the limits of the ISO-TP protocol, my own cars ECU was purposly bricked (Put into a bootloader state), and using Veidmao (Daimler's ECU Flashing tool that supports the J2534 API), was successfully restored. As shown in the below

image, Vediamo sends thousands of 4096 byte ISO-TP payloads to the adapter to send to the ECU over ISO-TP, which contain the data to be flashed onto the ECU. Also transmitted are some smaller payloads, which keep the ECU in a flash diagnostic session and to control other ECUs in the vehicle.

Vediamo tells the other ECUs in the vehicle to stop sending normal CAN messages, and go silent. This allows the full 500kbps of the CAN Network to be utilized for the firmware update, which in turn reduces the chances of lost CAN Packets and makes the flashing process faster.

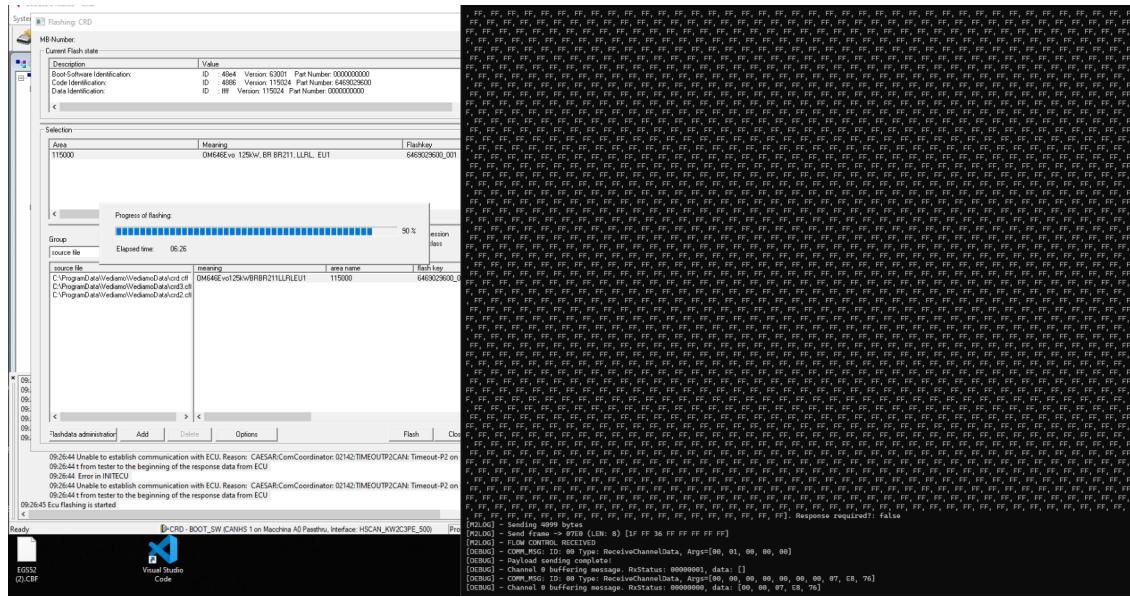


Figure 4.2: Vediamo utilizing the custom J2534 adapter to flash a ECU

From running these 2 applications with the custom Passthru driver, it is possible to conclude that for the 2 implemented protocols (CAN and ISO-TP), the adapter is fully function and therefore is a success.

4.2 Diagnostic application

In this section, parts of the diagnostic application will be tested on multiple vehicles, as well as the impact it has on consumer car diagnostics. All tests and findings mentioned in this part of the report will be based on the following two vehicles:

For testing, two vehicles were used in order to verify both UDS and KWP2000 function:

1. Mercedes W203 (2006 C class) - For KWP2000
2. Mercedes W246 (2018 B class) - For UDS

4.2.1 Automated ISO-TP Scanner

After completing a scan on both cars, the following ISO-TP endpoints were discovered:

ISO-TP configuration				Diagnostic info			
Send addr.	Receive addr.	BS	ST (ms)	KWP2000	UDS	ECU name	ECU description
0x0563	0x04E3	8	40	Yes	No	SAM-H	Front SAM
0x05B4	0x04F4	8	40	Yes	No	KOMBI	Cluster
0x0662	0x04E2	8	40	Yes	No	SAM-V	Rear SAM
0x0667	0x04E7	8	40	Yes	No	DBE	Overhead panel
0x06C8	0x04E8	8	40	Yes	No	TVL	Front left door
0x06CA	0x04EA	8	40	Yes	No	TVR	Front right door
0x0749	0x04E9	8	40	Yes	No	THL	Rear left door
0x074B	0x04EB	8	40	Yes	No	THR	Rear right door
0x0778	0x0789	8	10	Yes	No	EWM	Gear selector module
0x07E0	0x07E8	8	10	Yes	No	MS	Engine control module
0x07E1	0x07E9	8	10	Yes	No	GS	Gearbox control module
0x0784	0x0785	8	10	Yes	No	BS	ESP control module

Table 4.1: Automated scan results on the Mercedes W203

ISO-TP configuration				Diagnostic info	
Send addr.	Receive addr.	BS	ST (ms)	KWP2000	UDS
0x06C9	0x0459	8	20	No	No
0x0642	0x0488	8	20	No	Yes
0x06F3	0x04DE	8	20	No	Yes
0x07E3	0x07EB	8	20	No	Yes
0x0753	0x04EA	8	10	No	Yes
0x064A	0x0489	8	20	No	Yes
0x0703	0x04E0	8	20	No	Yes
0x06A3	0x04D4	8	20	No	Yes
0x07E5	0x07ED	8	20	No	Yes
0x068B	0x04D1	8	20	No	Yes
0x07F1	0x07F9	8	20	No	Yes
0x0632	0x0486	0	0	No	Yes
0x06A9	0x0455	8	20	No	No
0x078A	0x04B1	8	20	No	Yes
0x0691	0x0452	8	20	No	No
0x065A	0x048B	8	20	No	Yes
0x0622	0x0484	8	20	No	Yes
0x06B2	0x0496	8	20	No	Yes
0x0612	0x0482	8	20	No	No
0x0652	0x048A	8	20	No	Yes
0x06E2	0x06E1	8	20	No	No
0x0743	0x04E8	8	10	No	Yes
0x06E1	0x06E2	8	20	No	No
0x063B	0x04C7	8	20	No	Yes
0x070B	0x04E1	8	20	No	Yes
0x0732	0x04A6	8	20	No	Yes
0x07E0	0x07E8	8	10	No	Yes
0x07E1	0x07E9	8	20	No	Yes
0x0733	0x04E6	8	10	No	Yes

Table 4.2: Automated scan results on the Mercedes W246

As seen in table 4.1, the automated scanner only located diagnosable ECUs. the ECU name is found by cross referencing the part number in the JSON with google, which gives the name of the ECU. This unfortunately is not possible with UDS, so table 4.2 does not show ECU identification data.

One interesting thing about these two results is that in the W203, it appears the OBD-II port is properly firewalled, only allowing diagnosable ISO-TP addresses to be located, without also locating other ISO-TP endpoints such as Radio to Instrument cluster communication. This however is not true for the W246, as it located multiple ISO-TP endpoints in the vehicle which don't support either UDS or KWP2000, meaning these are likely ECU to ECU ISO-TP endpoints.

4.2.2 Diagnostic session mode (JSON)

The main thing users would be using this for is to get a clear description of error codes stored on an ECU, and to clear the error codes. Here, DTC errors are shown using data provided by JSON, as well as the description of each error, the status of each error, and additional freeze frame data about each error when the user selects an error in the top table.

ECU Error view			
Error	Description	State	MIL on
2054	Fuel temperature sensor-HP Pump-Electrical fault	Permanent	NO
2035	Engine coolant temperature sensor-Engine Bloc-Electrical fault	Permanent	NO
2086	Air pressure P1 sensor-In AMF-Electrical fault	Permanent	NO
23F4	VGT actuator-none-Driver fault	Permanent	NO
2379	Oil temperature sensor-Engine Block-Electrical fault	Permanent	NO
236A	EGR Cooler by pass actuator -none-electrical fault	Permanent	NO
206D	Inlet Air temperature sensor-In AMF-Electrical fault	Permanent	NO
22A5	Exhaust temperature-sensor before turbine-Electrical fault	Permanent	NO
20D3	EGR actuator-none-Driver fault	Permanent	NO
20E7	Inlet Metering Valve-HP Pump-Driver fault	Permanent	NO
201A	Boost pressure sensor-none-Electrical fault	Permanent	NO
22A0	High Pressure valve-Rail-Electrical fault	Permanent	NO
227C	SMIDP valve actuator-none-Driver fault	Permanent	NO

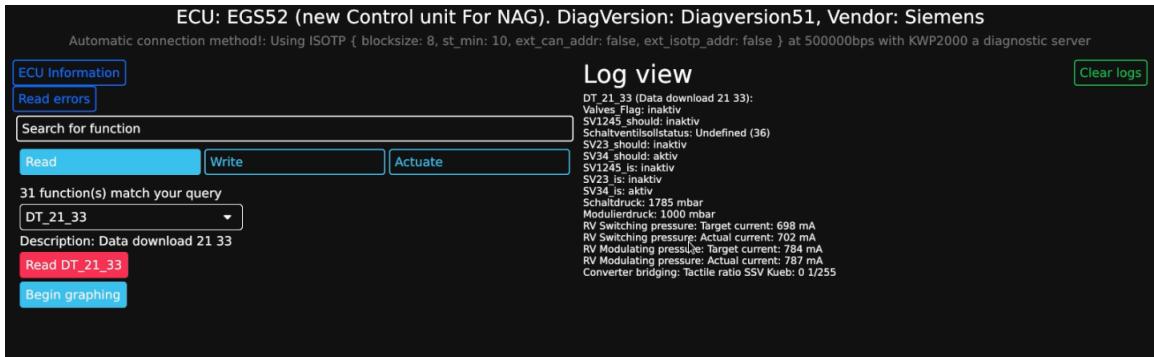
Freeze frame data

Engine Speed	0 rpm
U-Daten 1. Auftreten	0
Battery Voltage	11.921568 V
Boost Pressure Actuator duty cycle	49.80392 %
Boost Pressure	0.9803922 bar
Inlet Air Temperature	20 °C
Total Fuel Demand	0 mg/Hub
Engine state	SMC_STOPPED
Total Distance	0 Km

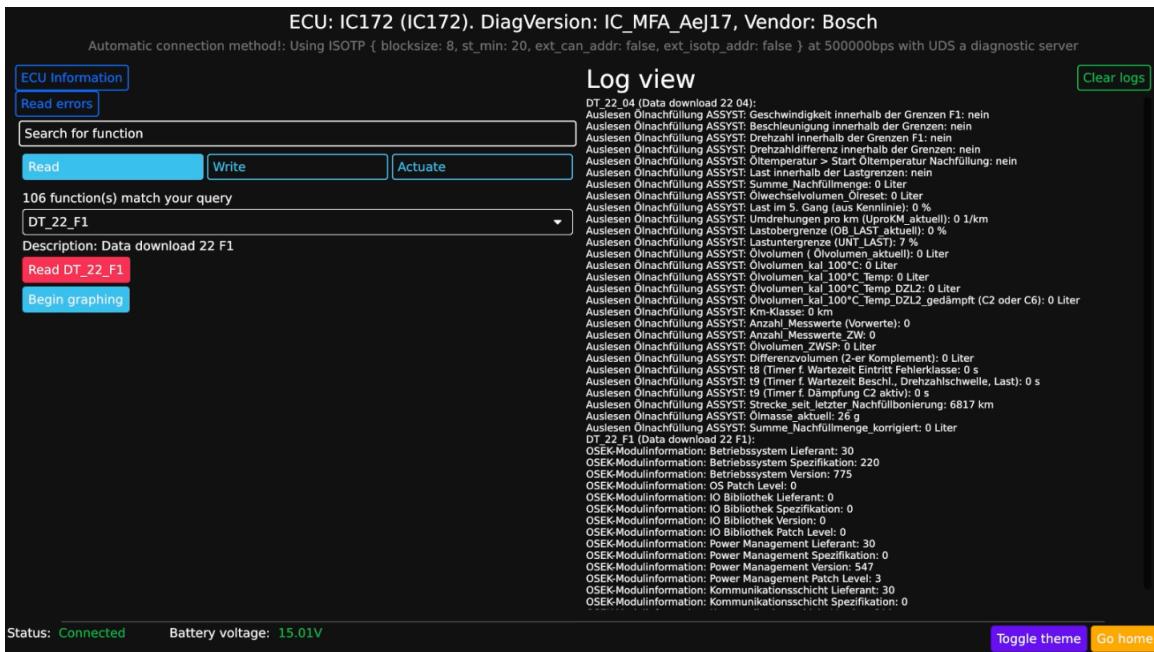
When pressing the 'clear errors' button, the ECU successfully clears all these errors, the JSON session re-reads the DTCs on the ECU immediately after clearing to verify that the errors have actually been cleared.

ECU Error view			
Read errors	Clear errors	No DTCs	Back

To prove that this works with multiple ECUs that are not necessarily engine ECUs, here is OpenVehicleDiag communicating with EGS52. This is a transmission controller for the 722.6 series gearboxes by Mercedes. In this picture, OpenVehicleDiag is connected to the ECU and reading data from the ECU about solenoid pressures. This data can only normally be retrieved using Daimler's own diagnostic software.



Lastly, here is the JSON sesison working with the Mercedes W246. In the below image, the JSON session is being used to talk to the vehicles instrument cluster to read various data:

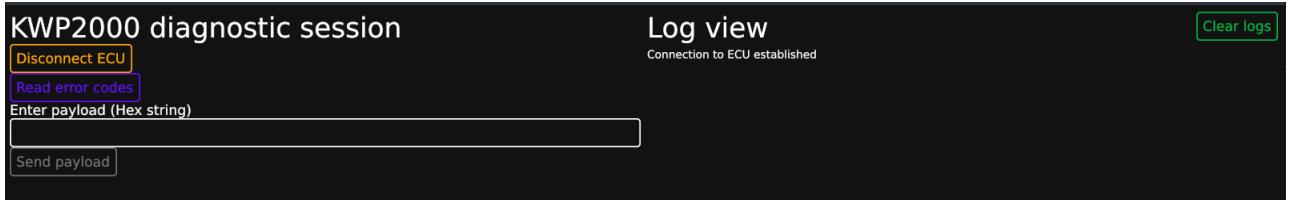


This proves that both KWP2000 and UDS have been correctly implemented within OpenVehicleDiag, and work reliably to do ECU diagnostics on both older and modern vehicles without any proprietary software.

4.2.3 Generic diagnostic session

As mentioned in section 3.5, a lot of features were not discussed in their implementation. One of those features was the Generic diagnostic session. This is launched within the application by using the results of an ECU detection scan (Or entering manual ISO-TP details). This

allows for basic diagnostics, but still being able to read and clear DTCs from ECUs that cannot normally be done under OBD-II. This works by opening up a generic KWP2000 or UDS session based on the output of the ECU scan results:



In this interface, the user has to manually enter commands to the ECU in hex form (However there is a button to read and clear DTCs). When reading DTCs in this mode, only the name of the error and state of the DTC can be read, the description of the error is unknown:



4.3 Summary

As shown in this chapter, everything mentioned in section 1 has been achieved, and is working reliably enough for it to be used in vehicles. Since a lot of OpenVehicleDiag's UI features are dynamic, a lot of the user interface has not been demonstrated in this chapter, just the main points. However, there is a video¹ that shows the entire interface and feature-set in use. There are also additional images of other parts of OpenVehicleDiag's user interface located at A.0.2.

The JSON diagnostic session results show that the decoder functions in JSON Schema code work perfectly as well, being able to decode Enums, Booleans, Integers and Strings from ECU response message byte streams, and nicely present them to the end user in a way which they can understand. The only thing that could not be tested here is verifying the decoder works with different endiannes. This is because the vast majority of ECUs utilize Big-Endian.

¹https://youtube.com/_k-dWdNRVr0

Chapter 5

Conclusions and Future Work

5.1 Conclusions

With the initial goal of this project being to attempt to create an open source alternative to commercial / proprietary diagnostic software for vehicle ECUs, as well as an open storage format for ECU diagnostic data, the initial results as outlined in this report have shown that it is indeed possible.

The Passthru library has shown that it is indeed possible to write firmware for a custom Passthru compliant adapter, and utilize it with applications that utilize the API. Although K-Line and J1850 based protocols were not implemented as part of this project, the testing with just CAN and ISO-TP protocols proves that the M2 is fully compliant with the API protocol, and more importantly, is stable.

With OpenVehicleDiag, the application has proven that a cross-platform ECU diagnostics platform can be achieved, and more importantly, be done in a way which can theoretically work with any vehicle, without any prior knowledge of the vehicle (For basic diagnostics with OBD-II, KWP2000 and UDS). Combining OpenVehicleDiag with the JSON Schema can improve the diagnostic capability of the application significantly, allowing for some functions only found in OEM tools.

The JSON Schema has shown that there can be an easy accessible replacement for the ODX specification. However, there are things that are missing, which would need to be added later on in order for it to be a true competitor to the ODX specification. In diagnostic tools, there are tests which are defined as a list of operations to perform on the ECU

5.2 Future work

With the success of this project so far, this project will be continued long into the future. This is especially relevant at the time of writing with movements such as 'Right to repair' gaining traction in the United States and European Union, as such an application would open the possibility for consumers to diagnose ECUs themselves with a simple, easy to use application.

The Macchina M2 Passthru library has also been successful, proving that expensive proprietary adapters are not necessarily needed for car diagnostics. At the time of writing, work is already underway¹ to create a unified repository based on the code that was written for the M2 adapter, and port it to the Macchina A0. The A0 is a cheaper and smaller adapter which only supports CAN, and is based on the ESP32 micro-controller running FreeRTOS. This means that it is only good for more modern vehicles built around 2006 and newer. Eventually, there

¹<https://github.com/rnd-ash/Macchina-J2534>

are also plans to create a D-PDU (2.2.1) driver for both the M2 and A0, and add D-PDU functionality to OpenVehicleDiag itself, which would widen the list of supported protocols and existing adapter that can be utilized.

OpenVehicleDiag at the time of writing this now has initial support for SocketCAN [kernel.org (n.d.)] on Linux based OS's, meaning that in theory it is possible to use OpenVehicleDiag on nothing more than a raspberry Pi and a \$5 CAN Shield based on the MCP2515 CAN Transceiver (Assuming the target vehicle supports CAN). This will hopefully encourage more users to utilize OpenVehicleDiag due just how low cost this solution will be compared to purchasing other adapter.

Stated in 1, it was mentioned that ECU Flashing will not be a part of OpenVehicleDiag, however, eventually it might be a good idea to add ECU firmware dumping to OpenVehicleDiag, which would allow more advanced users who want to reverse engineer their own ECUs to dump the firmware of the ECU and load it up in a disassembler such as Ghidra to understand how it operates. This would aid in the development of open source ECU firmware, such as projects like Speeduino². This is because every engine operates slightly differently, and also modern engine ECUs broadcast tons of CAN Data to other ECUs in a vehicle. Reverse engineering ECU firmware will allow such projects to cater for more engines and vehicles.

Lastly, one area of future work would be to create a Graphical user interface for users to create the JSON format used for ECU diagnostics, using simple 'drag and drop' blocks. This would vastly simplify the process of creating ECU diagnostic data in JSON. This would be a direct parallel to Vector's ODXStudio, which allows for OEMs to create ODX diagnostic data using a graphical tool, and then for the data to be saved in an ODX XML file.

²<https://speeduino.com/home>

Chapter 6

Reflection

This project has given me loads of insights and understanding of how ECU diagnostic protocols function and how easy they actually are to implement once obtaining the correct knowledge on the specification. One interesting skill from this project is creating a user interface dynamically in pure code without any form of UI designer. When looking back at the original objectives of the project, the outcome was more than was originally planned, and with the community support received during the projects development, the project will live on and gain more features over time.

With regards to the JSON Schema for OpenVehicleDiag, this was probably the hardest component out of the three components developed for this project, since the ODX file specification is not well known. Therefore, creating my own parallel for ODX-D but in JSON opens the possibility for others to create diagnostic containers for ECUs without the need of advanced commercial tools.

The Passthru library creation using Macchina's M2 has shown that it is indeed possible to create an open source alternative to professional diagnostic adapters. Although difficult to implement, it was interesting to see how its possible to create a cross-platform of an API which was originally designed for older versions of Windows, utilizing certain cross-platform libraries in Rust.

References

Comer352L (2019), 'Sae j2534 definitions header'.

URL: <https://github.com/Comer352L/FreeSSM/blob/master/src/J2534.h>

DaimlerChrysler (2002), 'Keyword protocol 2000 requirements definition'.

URL: http://read.pudn.com/downloads554/ebook/2284613/KWP2000_release2_2.pdf

Drew Technologies, Inc (2003), 'Sae j2534 api reference'. J2534 API reference.

URL: http://read.pudn.com/downloads209/ebook/984970/PassThru_API-1.pdf

elmelectronics.com (2017), 'Elm327'.

URL: <https://www.elmelectronics.com/wp-content/uploads/2017/01/ELM327DS.pdf>

emotive de (2014a), 'Diagnoselayer und diagnosedienste'. ODX information webpage.

URL: https://www.emotive.de/wiki/index.php?title=Diagnoselayer_und_Diagnosedienste

emotive de (2014b), 'Odx basic structure'. ODX information webpage.

URL: https://www.emotive.de/wiki/index.php?title=ODX_Grundstruktur

ISO (1989), 'Road vehicles — diagnostic systems — requirements for interchange of digital information'.

URL: <https://www.iso.org/standard/16737.html>

ISO (2000), 'Road vehicles – diagnostic systems – keyword protocol 2000 – part 4: Requirements for emission-related systems'.

URL: <https://www.iso.org/standard/28826.html>

ISO (2006), 'Road vehicles - unified diagnostic services (uds) - specification and requirements'.

URL: [http://read.pudn.com/downloads191/doc/899044/ISO+14229+\(2006\).pdf](http://read.pudn.com/downloads191/doc/899044/ISO+14229+(2006).pdf)

kernel.org (n.d.), 'Readme file for the controller area network protocol family (aka socketcan)'.

URL: <https://www.kernel.org/doc/Documentation/networking/can.txt>

Nils Weiss, Sebastian Renner, Jürgen Mottok, Václav Matoušek (n.d.), 'Transport layer scanning for attack surface detection in vehicular networks'.

Softing (2013), 'D-pdu api software for communication interfaces'. D-PDU API reference.

URL: <https://www.slideshare.net/linhdoanbro/d-pduapi-usermanual>

Wikipedia (2021), 'Obd-ii pids'.

URL: https://en.wikipedia.org/wiki/OBD-II_PIDs

Appendix A

Screenshots and diagrams

A.0.1 Daimler Xentry software

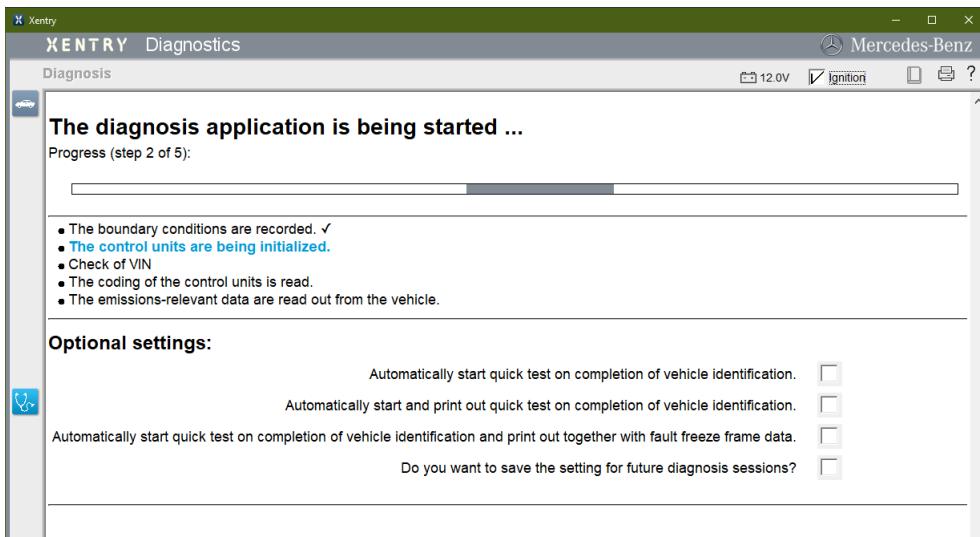


Figure A.1: Xentry Diagnostics establishing communication with all the ECUs within a vehicle. During this stage of diagnostics, Xentry is trying to locate all the ECUs on the vehicle, and checking what variation each ECU is in order to parse their diagnostic data correctly

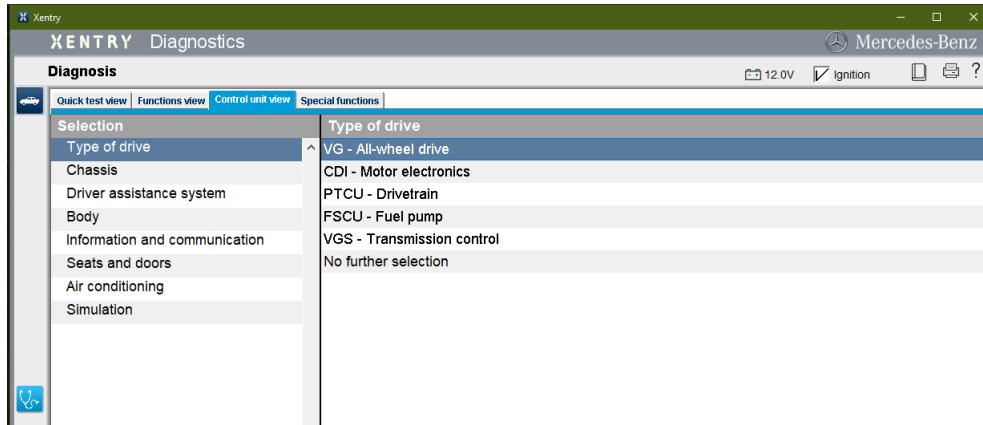


Figure A.2: Xentry diagnostics with a list of all possible ECUs in the vehicle to talk to, each in their own category

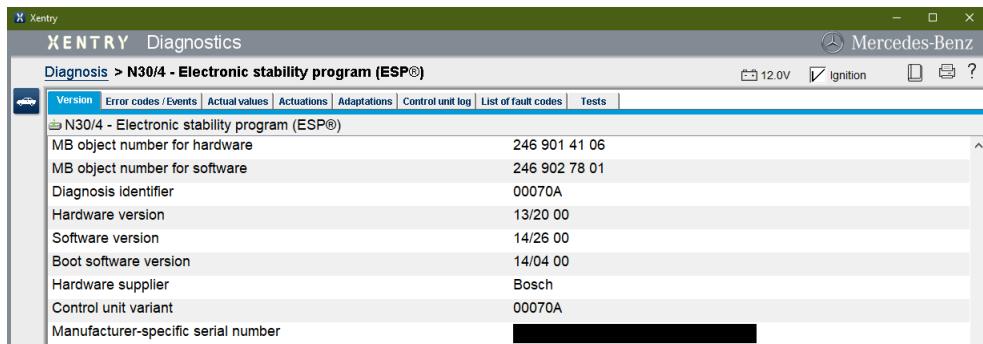


Figure A.3: Obtaining advanced data from the ECU in Xentry - Querying various attributes about the ESP ECU. The serial number of this part has been hidden.

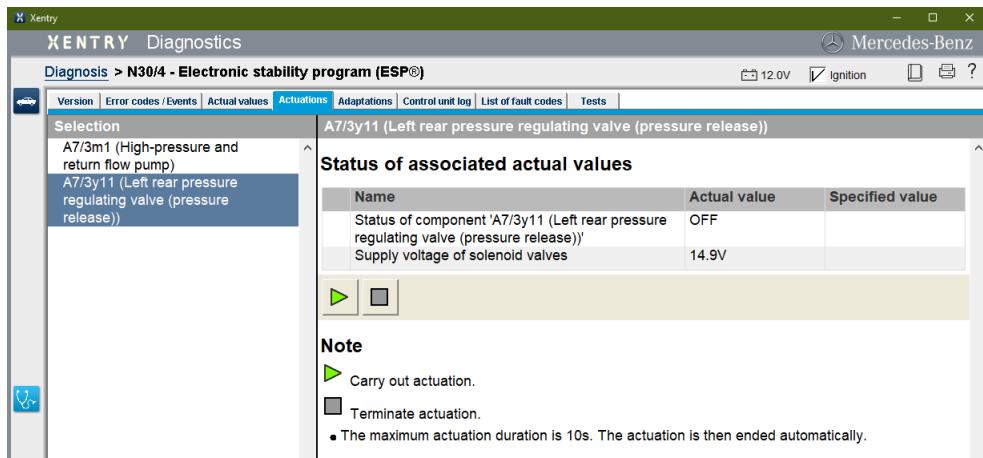


Figure A.4: Xentry showing a live 'actuation' value of certain items the ESP ECU controls

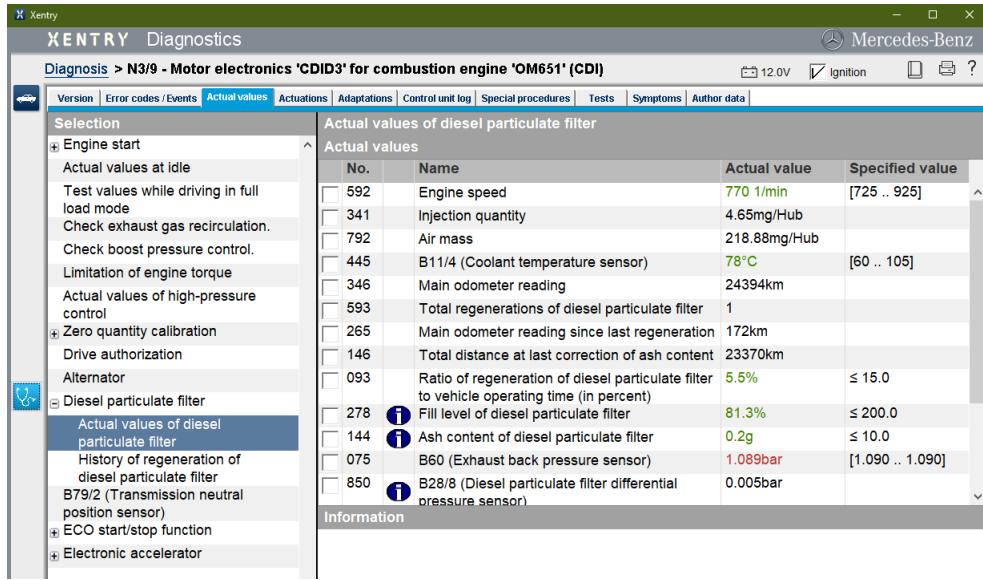


Figure A.5: Show Xentry obtaining real-time data from the CDI Engine ECU. The values in Green are within tolerance, and values in red are outside tolerance. Black values indicate no tolerances are specified for the value

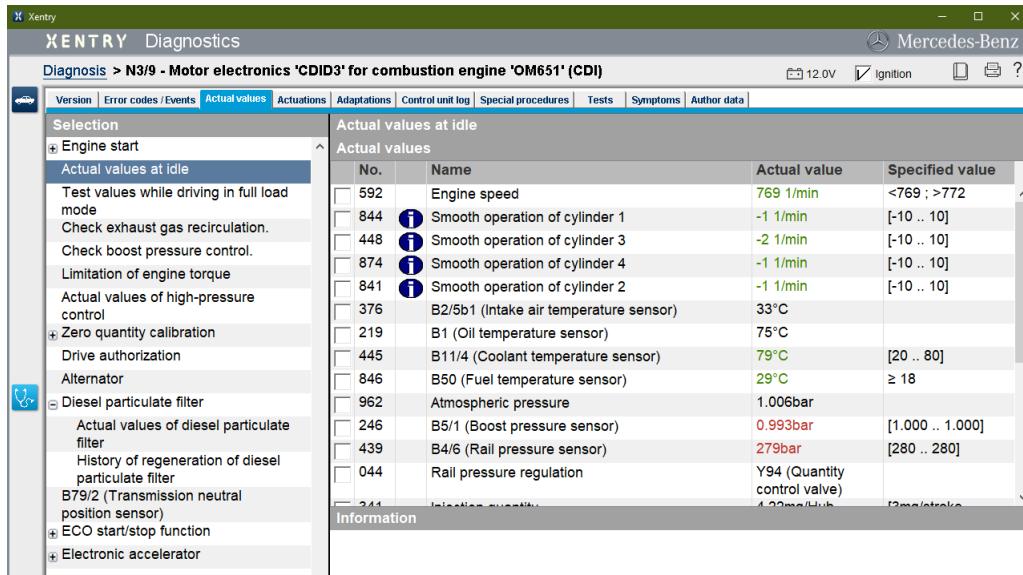


Figure A.6: Xentry showing advanced real-time data from the CDI Engine ECU. This allows for advanced analytics of how the engine is performing.

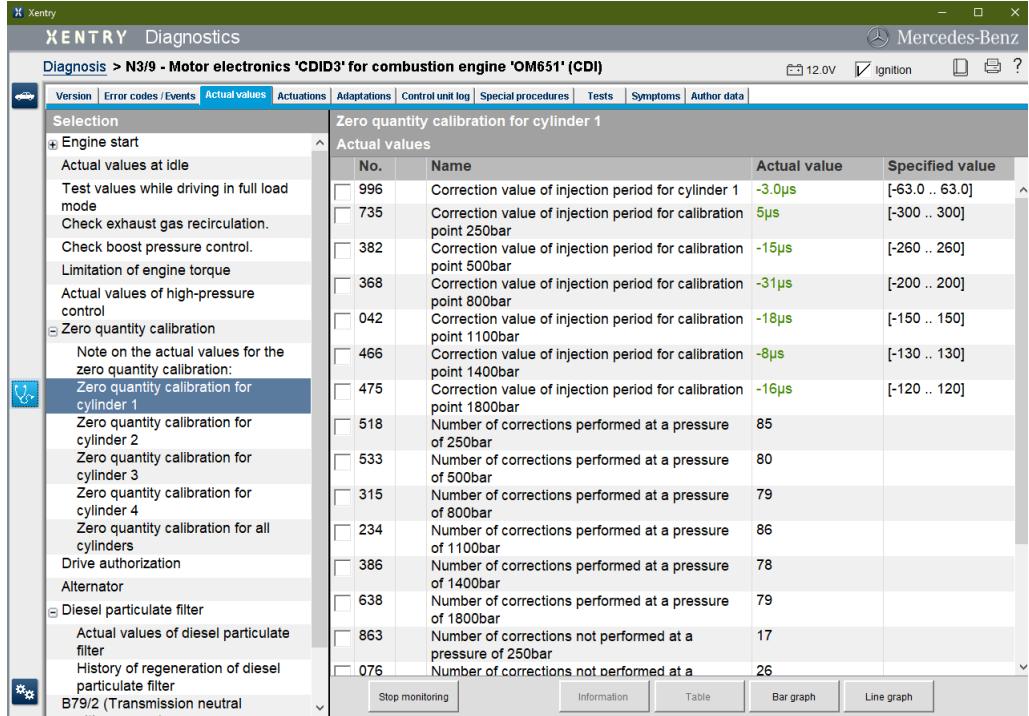


Figure A.7: More advanced real-time diagnostics with the CDI Engine ECU. This shows the injector calibration values for the number 1 cylinder

A.0.2 OpenVehicleDiag

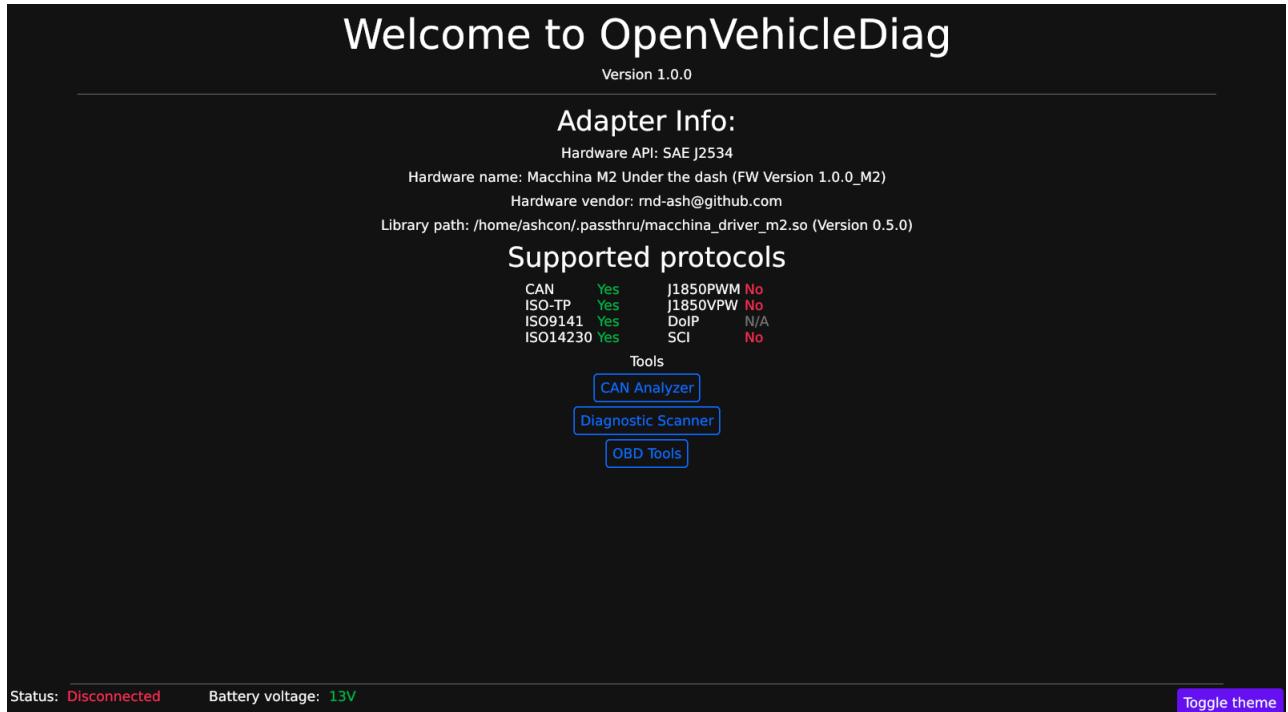


Figure A.8: OVD Home page (Dark theme)

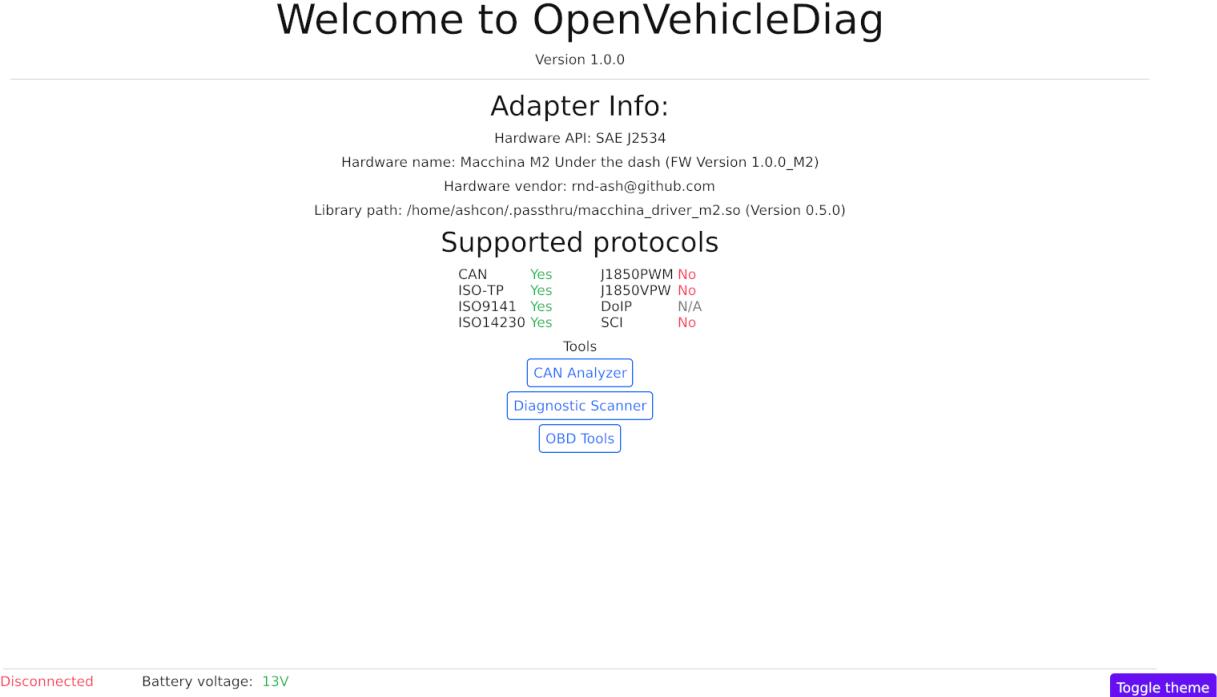


Figure A.9: OVD Home page (Light theme)

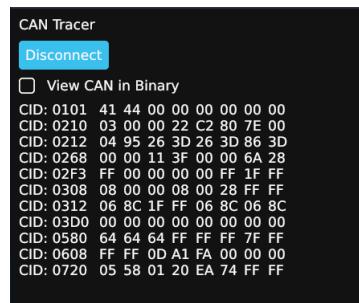


Figure A.10: OVD Can Scanner (Hex mode)

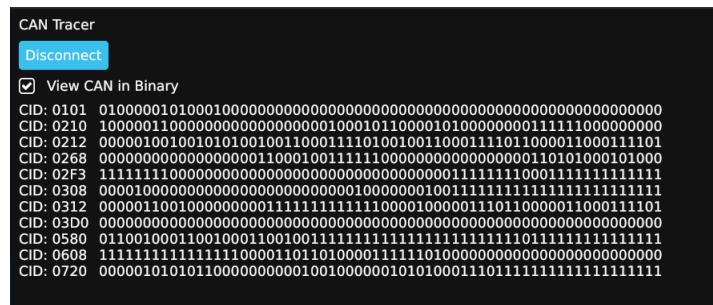


Figure A.11: OVD Can Scanner (Binary mode)

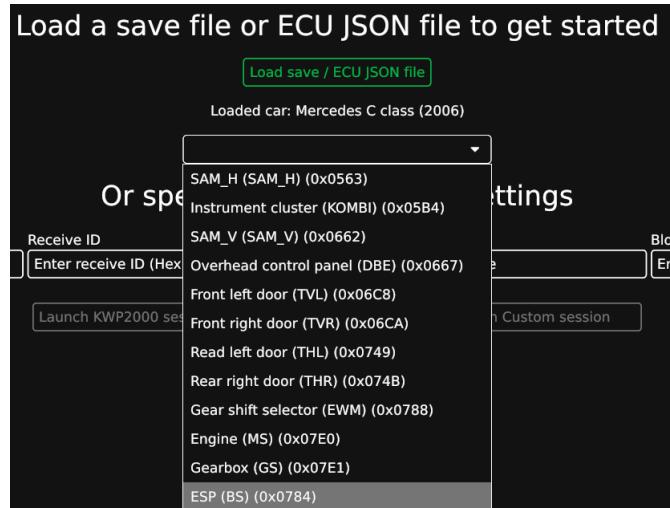


Figure A.12: Loading a ECU Scan save file in OVD

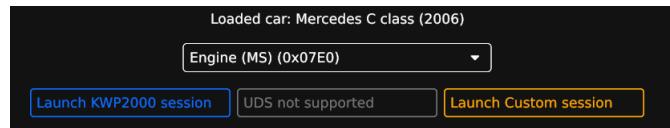


Figure A.13: Selected CRD ECU in OVD

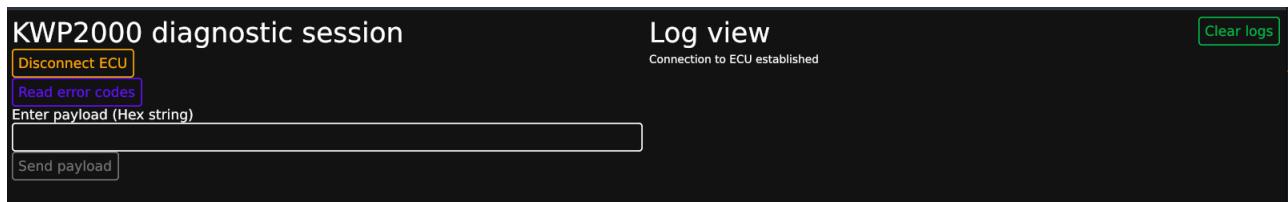


Figure A.14: OVD KWP2000 generic session - Home

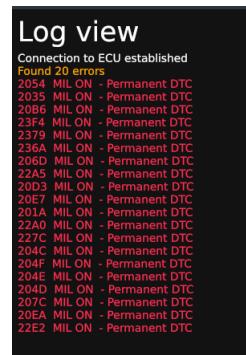


Figure A.15: OVD KWP2000 generic session - Scanning DTCs

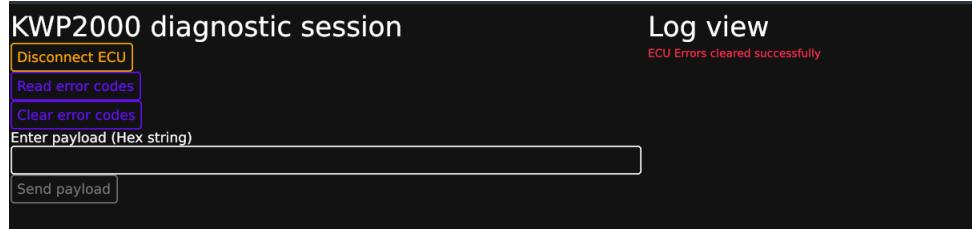


Figure A.16: OVD KWP2000 generic session - Clearing DTCs

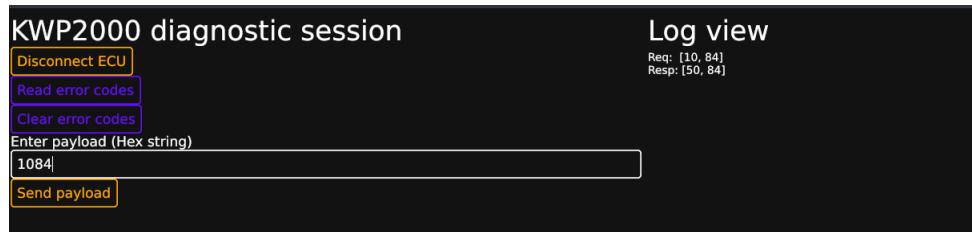


Figure A.17: OVD KWP2000 generic session - Sending valid manual payload

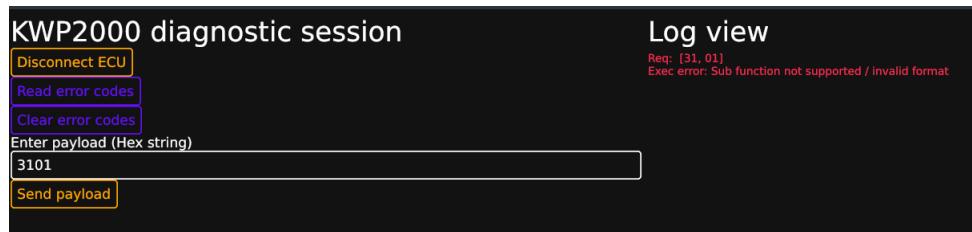


Figure A.18: OVD KWP2000 generic session - Sending invalid manual payload



Figure A.19: OVD Json session - Connected to CRD engine ECU

ECU Info view	
Name	CRD
Description	CRD.COMmon-Rail delphi
Software	CRD_Diag_86h
Manufacture	delphi
Part number	0044461540
Hardware version	0621
Software version	115024
Hardware build date (WW/YY)	21/06
Software build date (WW/YY)	50/11
Production date (DD/MM/YY)	15/10/07

Figure A.20: OVD Json session - ECU Info page

ECU Error view			
		Read errors	Clear errors
2054	Fuel temperature sensor-HP Pump-Electrical fault	Permanent	NO
2035	Engine coolant temperature sensor-Engine Bloc-Electrical fault	Permanent	NO
2086	Air pressure P1 sensor-In AMF-Electrical fault	Permanent	NO
23F4	VGT actuator-none-Driver fault	Permanent	NO
2379	Oil temperature sensor-Engine Block-Electrical fault	Permanent	NO
236A	EGR Cooler by pass actuator -none-electrical fault	Permanent	NO
206D	Inlet Air temperature sensor-In AMF-Electrical fault	Permanent	NO
22A5	Exhaust temperature-sensor before turbine-Electrical fault	Permanent	NO
20D3	EGR actuator-none-Driver fault	Permanent	NO
20E7	Inlet Metering Valve-HP Pump-Driver fault	Permanent	NO
201A	Boost pressure sensor-none-Electrical fault	Permanent	NO
22A0	High Pressure valve-Rail-Electrical fault	Permanent	NO
227C	SMID1 valve actuator-none-Driver fault	Permanent	NO

Figure A.21: OVD Json session - DTC page

ECU Error view			
Error	Description	State	MIL on
2054	Fuel temperature sensor-HP Pump-Electrical fault	Permanent	NO
2035	Engine coolant temperature sensor-Engine Bloc-Electrical fault	Permanent	NO
2086	Air pressure P1 sensor-In AMF-Electrical fault	Permanent	NO
2384	VGT actuator-none-Driver fault	Permanent	NO
2379	Oil temperature sensor-Engine Block-Electrical fault	Permanent	NO
236A	EGR Cooler by pass actuator -none-electrical fault	Permanent	NO
206D	Inlet Air temperature sensor-In AMF-Electrical fault	Permanent	NO
22A5	Exhaust temperature-sensor before turbine-Electrical fault	Permanent	NO
20D3	EGR actuator-none-Driver fault	Permanent	NO
20E7	Inlet Metering Valve-HP Pump-Driver fault	Permanent	NO
201A	Boost pressure sensor-none-Electrical fault	Permanent	NO
22A0	High Pressure valve-Rail-Electrical fault	Permanent	NO
227C	SWIBI valve actuator-none-Driver fault	Permanent	NO

Freeze frame data

Engine Speed	0 rpm
U-Daten 1. Auftreten	0
Battery Voltage	11.921568 V
Boost Pressure Actuator duty cycle	49.80392 %
Boost Pressure	0.9803922 bar
Inlet Air Temperature	20 °C
Total Fuel Demand	0 mg/Hub
Engine state	SMC_STOPPED
Total Distance	0 Km

Figure A.22: OVD Json session - DTC page with Freeze frame interpretation

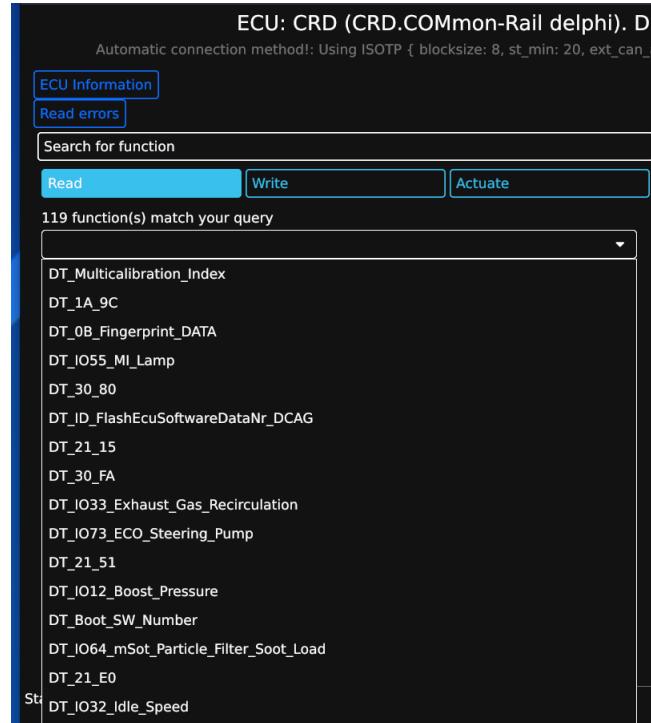


Figure A.23: OVD Json session - Selecting function to read data from

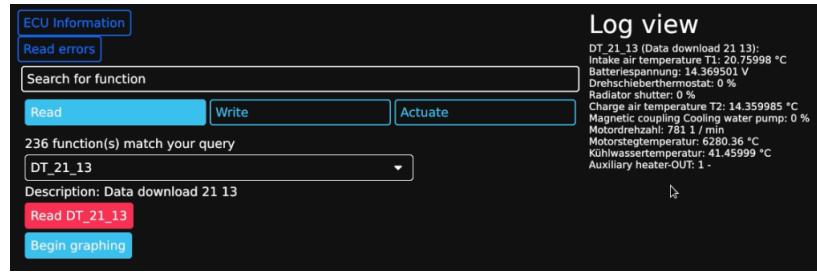


Figure A.24: OVD Json session - Data presentation

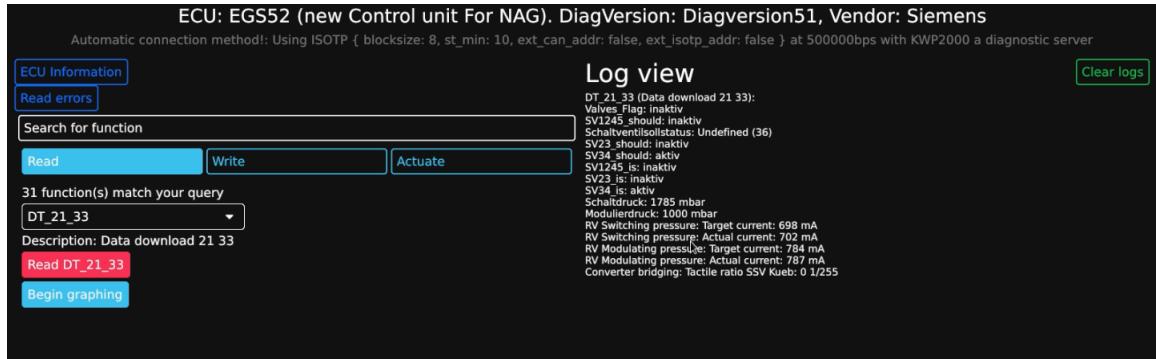


Figure A.25: OVD Json session - Reading data from EGS52 transmission ECU

Appendix B

Issues and resolutions

B.1 A full list of issues encountered during development of the M2 driver

B.1.1 Windows Serial API

The serialport-rs library has a flaw in Windows due to Windows's poorly documented Serial API. Unlike the UNIX world, In Windows, reading and writing to a COM Port are blocking by default, and will only return if the serial buffer is full. This means that if loads of data is being sent from the M2 module, the buffer would overflow before the driver has a chance to read all the data from within the buffer. This issue is still on-going, and can be tracked with an open issue with the serial library <https://gitlab.com/susurrus/serialport-rs/-/issues/95>

B.1.2 CAN Due library

During development with the CAN Interface on the M2, it transpired that the due_can library does not support mailbox reading (The ability to poll an individual Can Rx mailbox), even though the library does contain functions to do so.

After a discussion with Colin (The library Author), it was suggested that the M2 driver instead take advantage of the hardware interrupt feature of the CAN Mailboxes. This then led to the creation of 'custom_can.cpp' in which each mailbox now has its own Ring buffer, and whenever a CAN Frame is received by the mailbox, it triggers a hardware interrupt which allows the M2 to move the incoming CAN Frame to the mailboxes own ring buffer.

Appendix C

Code snippets and tables

C.1 List of SAE J2534 API functions

The following table is from [Drew Technologies, Inc (2003)]

Function name	Description
PassThruConnect	Establish a logical communication channel using the specified vehicle network protocol. Protocols supported can be extended by the Scan Tool vendor
PassThruDisconnect	Terminate an existing logical communication channel
PassthruReadMsgs	Receive network protocol messages from an existing logical communication channel
PassThruWriteMsgs	Transmit network protocol messages over an existing logical communication channel
PassThruStartPeriodicMsg	Transmit network protocol messages at the specified time interval over an existing logical communication channel
PassThruStopPeriodicMsg	Terminate the specified periodic message
PassThruStartMsgFilter	Transmit a network protocol filter that will selectively restrict or limit network protocol messages received by the User Application
PassThruStopMsgFilter	Terminate the specified message filter
PassThruSetProgrammingVoltage	Set the programmable voltage level on the specified J1962 connector pin
PassThruReadVersion	Retrieve the PassThru device firmware version, DLL version and API version information
PassThruGetLastError	Retrieve the text description for the last PassThru function that generated an error
PassThruIOCTL	General purpose I/O control function for retrieving and setting the various network protocol timing related parameters. Can be extended by the Scan Tool vendor to provide tool specific functionality

C.2 A full list of driver message types

For this table, the **Sender** column indicates which endpoint sent the first message containing this message type. The other end has to respond to the message unless stated otherwise.

Value	Definition (comm.h)	Sender	Description
0x00	MSG_NO_RESPONSE	Either	Identifies a message where no acknowledgment response is required from either the driver or firmware
0x01	MSG_LOG	Adapter	Identifies a log message from the adapter firmware. The payload bytes of this message shall me an ASCII String, which the PC driver will log
0x02	MSG_OPEN_CHANNEL	Driver	This message indicates the driver has requested that the adapter opens a logical communication channel on one of the OBD-II interfaces. The args of this message include the request Channel ID, the interface type, baud rate, and any additional arguments that are interface specific (EG: ISO-TP interface requires an additional argument to indicate if it is using extended addressing or not)
0x03	MSG_CLOSE_CHANNEL	Driver	This message indicates the driver wants the adapter to close a logical communication interface with the vehicle that was opened with MSG_OPEN_CHANNEL
0x04	MSG_SET_CHAN_FILT	Driver	This message tells the adapter to set a filter on a channel. The message's parameters will contain the target channel ID, msg lengths and filter bytes
0x05	MSG_REM_CHAN_FILT	Driver	This message tells the adapter to destroy a filter on a specified channel. The parameters will contain both the filter ID and channel ID
0x06	MSG_TX_CHAN_DATA	Driver	This message is sent to the adapter when data is to be transmitted to the vehicle. The parameters of this message will contain the channel ID as well as the raw bytes of the data to be sent to the vehicle.
0x07	MSG_RX_CHAN_DATA	Adapter	This message is sent to the driver from the adapter whenever the adapter has received any data on any active interface channels. Upon receiving this message, the drive will buffer the data in PASSTHRU_MSG structures and send them back to the user application when PassthruReadMsgs is called for the receiving channel.
0x08	MSG_READ_BATT	Driver	Requests the adapter to read the batter voltage on pin 16 of the OBD-II port. The response message args are formatted as a Big endian unsigned integer representing the voltage in millivolts
0x09	MSG_IOCTL_SET	Driver	Requests the adapter to set a specified IOCTL parameter for a specified channel to a specified value
0x10	MSG_IOCTL_GET	Driver	Requests the adapter to return a specified IOCTL parameter value for a specified channel
0xAA	MSG_STATUS	Driver	This message is sent to the adapter when the driver first loads or when the driver is unloaded, and is used to tell the adapter to either get ready to receive messages (On driver load), or to destroy all channels and reset to its default state (On driver unload)
0xAB	MSG_GET_FW_VERSION	Driver	Requests the adapter shall return a response message with an ASCII String in the parameters section which represents the current firmware version running on the adapter.
0xFF	MSG_TEST	Driver	Only used in tests and test compiled firmware. The adapter shall echo back this message with whatever data was received.

C.3 List of KWP2000 and UDS Services

SID: Service ID

SID Hex	Service name		Description
10	KWP2000	UDS	
10	StartDiagnosticSession	DiagnosticSessionControl	Used to control which diagnostic session the ECU is in. A default session is what the ECU is in after power on, and prohibits certain commands. Extended diagnostic session can be used to execute all commands. Flash session mode is typically used during firmware updates.
11		ECUReset	Used to tell the ECU to reset with a specified reset method. Once the ECU is reset, it will be back in a default session
14		ClearDiagnosticInfo	Used to clear DTCs from the ECU. A mask is typically provided, and DTCs which match the mask are cleared from the ECU.
17	ReadStatusOfDTC		Used to retrieve freeze frame data from the ECU about a given DTC code.
18	ReadDTCByStatus		Used to retrieve DTCs from the ECU by a provided status flag.
19		ReadDTCInformation	Used to retrieve DTCs from the vehicle by either status, group or mask.
1A	ReadECUIdentification		Requests identification data from the ECU.
21	ReadDataByLocalIdentifier		Reads data from the ECU's memory by a Local ID. A local ID is used to define what kind of data to retrieve. Multiple Local IDs can be provided within the same request message. Each local ID corresponds to a attribute in the ECU such as VIN number or OS version.
22		ReadDataByIdentifier	Used to request blocks of data from the ECU. The ID's are the same as what is used by WriteDataByIdentifier.
23		ReadMemoryByAddress	Used to retrieve data from the ECU's memory given a raw memory address.
24		ReadScalingDataByIdentifier	Same as ReadDataByIdentifier, except the ECU shall respond with a scaling data type (EG: Integer or Float), along with the algorithm used to convert it from raw to parsed.
27		SecurityAccess	Used to either generate a seed key, or to input a response to the ECU. If the response is valid to the provided seed key, the ECU will allow writing and reading of protected memory areas.
28	DisableNormalMsgTransmission		Used to disable normal ECU communication with other ECUs, leaving only diagnostic traffic untouched. This is typically sent to all ECUs in a vehicle prior to flashing a ECU in the vehicle as it allows for more bandwidth on the vehicles communication networks to be used for flashing.
28		CommunicationControl	Works in the same way as KWP2000's DisableNormalMsgTransmission and EnableNormalMsgTransmission commands.
29	EnableNormalMsgTransmission		used to re-enable normal ECU communication after they have been disabled with DisableNormalMsgTransmission
2C		DynamicallyDefineDataIdentifier	Used to dynamically create a new Local Identifier if none exists on the ECU already.

2E	WriteDataByIdentifier		Used to write blocks of data to the ECU. The ID's are the same as what is used by ReadDataByIdentifier
2F	IoctlByIdentifier		Used to control Input and output of the ECU for a specific component attached to the ECU. For example, the tester can use this function to manually actuate components on the ECU, then once done, return control back to the ECU.
30	IoctlByIdentifier		Used to control Input and output of the ECU for a specific component attached to the ECU. For example, the tester can use this function to manually actuate components on the ECU, then once done, return control back to the ECU.
31	StartRoutineByLocalIdentifier		Used to start execution of a test or routine in the ECU's memory. These routines will run a sequence of operations on the ECUs components. The test will not stop on the ECUs own accord, instead StopRoutineByLocalIdentifier must be used to stop the routine.
31	RoutineControl		This function combines the functionalities found in KWP2000's StopRoutineByLocalIdentifier, StartRoutineByLocalIdentifier and RequestRoutineResultsBylocalIdentifier commands.
32	StopRoutineByLocalIdentifier		Used to stop or terminate a routine that was started with StartRoutineByLocalIdentifier.
33	RequestRoutineResultsBylocalIdentifier		Used to retrieve the results of a routine execution that was started with StartRoutineByLocalIdentifier and stopped with StopRoutineByLocalIdentifier.
34	RequestDownload		Used to start a transfer between the tester and ECU. It is typically used in Flash routines. This function provides the memory address and compression type and uncompressed size of the data to be download
35	RequestUpload		Used to start a data transfer between the ECU and Tester. This function provides the memory address and compression type and uncompressed size of the data to be uploaded.
36	TransferData		Used to transfer data between the tester and ECU. The direction of data transfer would have been determined depending on if RequestDownload or RequestUpload was first executed.
37	RequestTransferExit		This is used to indicate that a data transfer has been completed.
3B	WriteDataByLocalIdentifier		Used to write data to the ECU given a specified Local ID. Each local ID corresponds to a attribute in the ECU such as VIN number or OS version.
3D	WriteMemoryByAddress		Used to write data to the ECU's memory given a raw memory address and content.
3E	TesterPresent		Sent periodically to the ECU in order to keep the ECU in a non-default diagnostic session.
83		AccessTimingParameter	Used to read and modify the default timing parameters utilizes for the communication link during a diagnostic session.

84		SecuredDataTransmission	Used to transfer data between the tester and ECU in a secure fashion, which is protected from attacks from third parties.
85		ControlDTCSetting	Used to enable or disable the logging of DTCs on the ECU.
86		ResponseOnEvent	Used to tell the ECU to either start or stop transmission of response messages on a specified event type, during a given window. An event can be anything which the ECU has access to such as timer interrupts, fault settings etc.
87		LinkControl	This is used in order to modify the baud rate of the transport protocol the ECU is utilizing during the diagnostic session.

C.4 Extract of OVD's JSON (EGS52) from CBFParser

The following is an extract from the JSON created by CBFParser for EGS52 (722.6 transmission TCM). All Strings here have been converted from German to English for easier understanding.

```

1 {
2   "name": "EGS52",
3   "description": "new Control unit For NAG",
4   "variants": [
5     {
6       "name": "Diagversion02",
7       "description": "EGS52-Diagversion02",
8       "patterns": [
9         {
10           "vendor": "Siemens",
11           "vendor_id": 4294935042
12         },
13         {
14           "vendor": "Siemens",
15           "vendor_id": 514
16         }
17       ],
18       "errors": [
19         {
20           "error_name": "P2000",
21           "summary": "",
22           "description": "58: Control unit EGS (Test internal Watchdog)",
23           "envs": [
24             {
25               "name": "EPS_Error detection since Init",
26               "unit": "",
27               "start_bit": 48,
28               "length_bits": 8,
29               "byte_order": "BigEndian",
30               "data_format": {
31                 "Bool": {
32                   "pos_name": "detected since Init",
33                   "neg_name": "restored out Storage"
34                 }
35               }
36             },
37             {
38               "name": "EPS_Error_status_internally",
39               "unit": "",
40               "start_bit": 48,
41               "length_bits": 8,
42               "byte_order": "BigEndian",
43               "data_format": "HexDump"
44             },
45             {
46               "name": "EPS_time_after_Reset, last Occur",
47               "unit": "sec",
48               "start_bit": 248,
49               "length_bits": 16,
50               "byte_order": "BigEndian",
51               "data_format": {
52                 "Linear": {
53                   "multiplier": 1.0,
54                   "offset": 0.0
55                 }
56               }
57             }
58           ]
59         }
60       ]
61     }
62   ]
63 }
```

```

56         }
57     }
58   ]
59 }
60 ],
61 "downloads": [
62 {
63   "name": "DT_21_33",
64   "description": "Data download 21_33",
65   "payload": "2133",
66   "output_params": [
67     {
68       "name": "Valves_Flag",
69       "unit": "",
70       "start_bit": 16,
71       "length_bits": 1,
72       "byte_order": "BigEndian",
73       "data_format": {
74         "Bool": {
75           "pos.name": "Active",
76           "neg.name": "Inactive"
77         }
78       }
79     },
80     {
81       "name": "Should activate solenoid 1-2/4-5",
82       "unit": "",
83       "start_bit": 18,
84       "length_bits": 1,
85       "byte_order": "BigEndian",
86       "data_format": {
87         "Bool": {
88           "pos.name": "Active",
89           "neg.name": "Inactive"
90         }
91       }
92     },
93     {
94       "name": "Shift solenoid target status",
95       "unit": "",
96       "start_bit": 24,
97       "length_bits": 8,
98       "byte_order": "BigEndian",
99       "data_format": {
100         "Table": [
101           {
102             "name": "None",
103             "start": 0.0,
104             "end": 0.0
105           },
106           {
107             "name": "1-2/4-5",
108             "start": 1.0,
109             "end": 1.0
110           },
111           {
112             "name": "2-3",
113             "start": 2.0,
114             "end": 2.0
115           },
116           {
117             "name": "1-2/4-5 and 2-3",
118             "start": 3.0,
119             "end": 3.0
120           },
121           {
122             "name": "3-4",
123             "start": 4.0,
124             "end": 4.0
125           },
126           {
127             "name": "1-2/4-5 and 3-4",
128             "start": 5.0,
129             "end": 5.0
130           },
131           {
132             "name": "2-3 and 3-4",
133             "start": 6.0,
134             "end": 6.0
135           },
136           {
137             "name": "1-2/4-5 and 2-3 and 3-4",
138             "start": 7.0,
139             "end": 7.0
140           }
141         ]
142       }
143     },
144     {
145       "name": "Should activate solenoid 2-3",
146       "unit": "",
147       "start_bit": 25,
148       "length_bits": 1,
149       "byte_order": "BigEndian",
150       "data_format": {

```

```

151         "Bool": {
152             "pos.name": "Active",
153             "neg.name": "Inactive"
154         }
155     },
156     {
157         "name": "Should active solenoid 3-4",
158         "unit": "",
159         "start_bit": 26,
160         "length_bits": 1,
161         "byte_order": "BigEndian",
162         "data_format": {
163             "Bool": {
164                 "pos.name": "Active",
165                 "neg.name": "Inactive"
166             }
167         }
168     },
169     {
170         "name": "Solenoid 1-2/4-5",
171         "unit": "",
172         "start_bit": 27,
173         "length_bits": 1,
174         "byte_order": "BigEndian",
175         "data_format": {
176             "Bool": {
177                 "pos.name": "Active",
178                 "neg.name": "Inactive"
179             }
180         }
181     },
182     {
183         "name": "Solenoid 2-3",
184         "unit": "",
185         "start_bit": 28,
186         "length_bits": 1,
187         "byte_order": "BigEndian",
188         "data_format": {
189             "Bool": {
190                 "pos.name": "Active",
191                 "neg.name": "Inactive"
192             }
193         }
194     },
195     {
196         "name": "Solenoid 3-4",
197         "unit": "",
198         "start_bit": 29,
199         "length_bits": 1,
200         "byte_order": "BigEndian",
201         "data_format": {
202             "Bool": {
203                 "pos.name": "Active",
204                 "neg.name": "Inactive"
205             }
206         }
207     },
208     {
209         "name": "Shift pressure",
210         "unit": "mbar",
211         "start_bit": 32,
212         "length_bits": 16,
213         "byte_order": "BigEndian",
214         "data_format": "Identical"
215     },
216     {
217         "name": "Modulating pressure",
218         "unit": "mbar",
219         "start_bit": 48,
220         "length_bits": 16,
221         "byte_order": "BigEndian",
222         "data_format": "Identical"
223     },
224     {
225         "name": "Switching pressure: Target current",
226         "unit": "mA",
227         "start_bit": 64,
228         "length_bits": 16,
229         "byte_order": "BigEndian",
230         "data_format": "Identical"
231     },
232     {
233         "name": "Switching pressure: Actual current",
234         "unit": "mA",
235         "start_bit": 80,
236         "length_bits": 16,
237         "byte_order": "BigEndian",
238         "data_format": "Identical"
239     },
240     {
241         "name": "Modulating pressure: Target current",
242         "unit": "mA",
243         "start_bit": 96,
244         "length_bits": 16,

```

```
246         "byte_order": "BigEndian",
247         "data_format": "Identical"
248     },
249     {
250         "name": "Modulating pressure: Actual current",
251         "unit": "mA",
252         "start_bit": 112,
253         "length_bits": 16,
254         "byte_order": "BigEndian",
255         "data_format": "Identical"
256     },
257     {
258         "name": "Torque converter target lockup",
259         "unit": "1/255",
260         "start_bit": 128,
261         "length_bits": 8,
262         "byte_order": "BigEndian",
263         "data_format": "Identical"
264     }
265   }
266 }
267 ]
268 }
```

Appendix D

A list of project repositories

The below repositories are where the code for this thesis is located at. Each repository has its own purpose.

- Macchina M2 Passthru driver - <https://github.com/rnd-ash/MacchinaM2-J2534-Rust/>
- OpenVehicleDiag application and CBFParser - <https://github.com/rnd-ash/OpenVehicleDiag/tree/v1.0.0>
- Common J2534 code (Rust) - <https://github.com/rnd-ash/J2534-Rust>

Appendix E

OpenVehicleDiag JSON Schema

The following has been adapted from the Json Schema document from the OpenVehicleDiag repository at release 1.0¹

Version 1.0 (31/03/2021)

This document outlines the JSON Specification which OpenVehicleDiag uses for ECU diagnostics. It is designed to be a simple, easy to understand replacement for ODX, and proprietary data formats such as Daimler' CBF and SMR-D data format.

JSON Root

Example

```
1 {
2   "name": "Awesome ECU",
3   "description": "My awesome engine ECU!",
4   "variants": [],
5   "connections": []
6 }
```

Properties

	Type	Description	Required
name	String	Name of the ECU	Yes
description	String	A brief description of the ECU	Yes
variants	Array	A list of ECU Variants (See below)	Yes
connections	Array	A list of connection methods for communicating with the ECU. See below	Yes

ECU Variant

An ECU Variant is used to identify a particular software version of an ECU. Since an ECU can get software updates over time, this is necessary as with certain software updates, an ECU can change or modify error code descriptions and also add or remove diagnostic routines.

Example

¹<https://github.com/rnd-ash/OpenVehicleDiag/blob/v1.0.0/SCHEMA.md>

```

1 {
2   "name": "SW_V_01",
3   "description": "My Awesome ECU software version 0.1",
4   "patterns": [],
5   "errors": [],
6   "adjustments": [],
7   "actuations": [],
8   "functions": [],
9   "downloads": []
10 }

```

Properties

	Type	Description	Required
name	String	A short version string of the ECU software version	Yes
description	String	Description of the ECU software version	Yes
patterns	Array	A list of pattern objects that are used to identify the hardware vendor which implements this variant of ECU software	Yes
errors	Array	A list of errors that this ECU variant could potentially throw	Yes
adjustments	Array	A list of services that can be executed on the ECU variant in order to permanently modify certain features of the ECU, such as setting a new idle RPM. It should be noted that these services do not require seed key access prior to execution	Yes
actuations	Array	A list of services that can be executed on the ECU variant in order to manipulate components temporarily during a diagnostic session	Yes
functions	Array	A list of services that can be executed on the ECU variant that have no user input or output. These can include things such as ECU Reset or modifying the diagnostic session type	Yes
downloads	Array	A list of services that can be executed on the ECU variant in order to read data from the ECU	Yes

Pattern

An ECU Pattern is used to identify which hardware vendor is responsible for implementing the parent variant software version, since its possible for 1 ECU software to be installed on multiple vendors' ECUs, such as Bosch, Siemens and Delphi.

Example

```

1 {
2   "vendor": "rnd-ash@github.com",
3   "vendor_id": 12345
4 }

```

Properties

	Type	Description	Required
vendor	String	The manufacture name of the ECU	Yes
vendor_id	Integer	The vendor ID pattern. Every vendor has a unique vendor_id for every ECU. This is a 2 or 4 byte integer retrieved with read_dcs_id or read_dcx_mmc_id with ECUs that utilize KWP2000, or read_vendor_id for ECUs that utilize UDS	Yes

Error

An error block represents a DTC (Diagnostic trouble code) which the ECU could potentially throw under certain circumstances to signify something is wrong.

Example

```

1 {
2   "error_name": "P2082-002",
3   "summary": "MAF implausible",
4   "description": "Mass airflow sensor is producing inconsistent readings",
5   "envs": []
6 }
```

Properties

	Type	Description	Required
error_name	String	The shorthand error code, usually in the SAE J2012 or ISO 15031-6 format	Yes
summary	String	The summary of what the error signifies	Yes
description	String	A more detailed description of the error	Yes
envs	Array	A list of parameters for interpreting data returned about the DTC using <code>get_error_status</code> . This essentially interprets the ECU specific freeze frame data, which is a list of sensor measurements that the ECU captures when the error is triggered, which can be useful for debugging the error.	No

Service

A service is used to describe an operation that can be executed on the ECU during a diagnostic session

Example

```

1 {
2   "name": "Read injector status",
3   "description": "Retrieves the injector quantity per stroke for all cylinders",
4   "payload": "22FB",
5   "input_params": [],
6   "output_params": []
7 }
```

Properties

	Type	Description	Required
name	String	The name of the service	Yes
description	String	A description of what the service does	Yes
payload	Hex string	The raw payload to be sent to the ECU	Yes
input_params	Array	A list of parameters that are used for interpreting user input for values to append to the ECU request payload	No
output_params	Array	A list of parameters that are used to decode the ECU's positive response message to the command	No

Parameter

A parameter is used to define the data format used for either input or output data that the ECU uses, as well as defining where in the ECU Request or response message the value is

located at.

Example

```

1   {
2     "name": "Supply voltage",
3     "description": "Supply voltage being measured by the ECU",
4     "unit": "V",
5     "start_bit": 32,
6     "length_bits": 8,
7     "byte_order": "BigEndian",
8     "data_format": "Identical",
9     "valid_bounds": {
10       "upper": 100.0,
11       "lower": 0.0
12     }
13   }

```

Properties

	Type	Description	Required
name	String	Name of the parameter	Yes
description	String	Description of the parameter	Yes
unit	String	Optional unit string, which will be appended to the output value when being displayed as a string	No
start_bit	Integer	The position in the ECU payload or response where this parameter is located at	Yes
length_bits	Integer	The length in bits of the parameter	Yes
byte_order	String	The byte order of the parameter. Allowed values are BigEndian or LittleEndian	Yes
data_format	Enum	Data format (see below) of the parameter	Yes
valid_bounds	JSON	Multi use. If the parameter is in the input_parameters section of the parent service, this field denotes the upper and lower bounds of the user input. If the parameter is in the parent services' output_parameters section, it is used for denoting the maximum and minimum y axis values for graphing	No

A list of valid data formats for Parameter

Below is a full list of all possible allowed entries in a Parameter.

Binary

Description: The output value is formatted as a binary string.

Example JSON

```
1   "data_format": "Binary,
```

Example parsing

```

1   INPUT: [0x20]
2   OUTPUT: "0b00100000"
3
4   INPUT: [0x20, 0xFF]
5   OUTPUT: "[0b00100000, 0b11111111]"

```

HexDump

Description: The output value is formatted as a hex string like array.

Example JSON

```
1 "data_format": "HexDump",
```

Example parsing

```
1 INPUT: [0x20, 0xFF, 0x00]
2 OUTPUT: "[0x20, 0xFF, 0x00]"
```

String

Description: The output value is formatted as a String using a specified encoding option. This uses lossless decoding, so if bytes don't map to specific characters, **?** will be displayed in its place.

Example JSON

```
1 "data_format": {
2   "String": "Utf8"
3 },
```

Example parsing

```
1 INPUT: [0x54, 0x65, 0x73, 0x74, 0x20, 0x6D, 0x65, 0x73, 0x73, 0x61, 0x67, 0x65]
2 OUTPUT: "Test message"
```

Notes For the string encoding option, 3 possible values are allowed:

- **ASCII** - The String is encoded as ASCII (1 byte per character)
- **Utf8** - The String is encoded as UTF-8 (1 byte per character)
- **Utf16** - The String is encoded as UTF-16 (2 bytes per character)

Bool

Description: The output value is formatted as a Boolean, where 0 is interpreted as False, and any other value is interpreted as True. If the fields **pos_name** and **neg_name** are present, then the strings in those fields will replace the default "True", "False" Strings.

Example JSON

```
1 "data_format": "Bool",
2
3 "data_format": {
4   "Bool": {
5     "pos_name": "This is positive",
6     "neg_name": "This is negative"
7   }
8 },
```

Example parsing

```
1 # First JSON example
2 INPUT: 0x01
3 OUTPUT: "True"
4
5 INPUT: 0x00
6 OUTPUT: "False"
7
```

```

8
9  # Second JSON example
10 INPUT: 0x01
11 OUTPUT: "This is positive"
12
13 INPUT: 0xFF
14 OUTPUT: "This is positive"
15
16 INPUT: 0x00
17 OUTPUT: "This is negative"

```

Table

Description: The output value is formatted as a String based on a table which represents an Enum. Each enum entry in the table has a defined start and end (inclusive) value. Any number between the start and end value are accepted into the enums variant. If no Enum could be found for a given input value, then "UNDEFINED" is returned (See example parsing below).

Example JSON

```

1 "data_format": {
2   "Table": [
3     {
4       "name": "This value is between 0 and 10",
5       "start": 0.0,
6       "end": 10.0
7     },
8     {
9       "name": "This value is only 11",
10      "start": 11.0,
11      "end": 11.0
12    },
13    {
14      "name": "This value is only 100",
15      "start": 100.0,
16      "end": 100.0
17    }
18  ]
19 },

```

Example parsing

```

1 INPUT: [0x00]
2 OUTPUT: "This value is between 0 and 10"
3
4 INPUT: [0x05]
5 OUTPUT: "This value is between 0 and 10"
6
7 INPUT: [0x64]
8 OUTPUT: "This value is only 100"
9
10 INPUT: [0xFF]
11 OUTPUT: "UNDEFINED(0xFF)"

```

Identical

Description: The output value is formatted as a number based on the raw value extracted from the parent parameter's bit range.

Example JSON

```
1 "data_format": "Identical"
```

Example parsing

```

1 INPUT: [0x00]
2 OUTPUT: "0"
3
4 INPUT: [0x05]
5 OUTPUT: "5"
6
7 INPUT: [0x64]
8 OUTPUT: "100"
9
10 INPUT: [0xFF, 0xFF]
11 OUTPUT: "65565"
```

Linear

Description: The output value is calculated using a simple $y = mx + c$ equation, where the "multiplier" field represents the m component of the equation, and the "offset" field represents the c component.

Example JSON

```

1 "data_format": {
2   "Linear": {
3     "multiplier": 0.125,
4     "offset": -40.0
5   }
6 },
```

Example parsing

```

1 INPUT: [0x00]
2 OUTPUT: "-40.0"
3
4 INPUT: [0x10]
5 OUTPUT: "-38.0"
6
7 INPUT: [0xFF]
8 OUTPUT: "-8.125"
```

ScaleLinear

This does not function in OpenVehicleDiag 1.0!

Description: The output value is calculated using a table of linear functions.

RatFunc

This does not function in OpenVehicleDiag 1.0!

Description: The output value is calculated using a rational function.

ScaleRatFunc

This does not function in OpenVehicleDiag 1.0!

Description: The output value is calculated using a table of rational functions.

TableInterpretation

This does not function in OpenVehicleDiag 1.0!

Description: The output value is calculated using defined interpolation.

Compucode

This does not function in OpenVehicleDiag 1.0!

Description: The output value is calculated using a Java virtual machine that runs bytecode of a class that implements the **I_CompCode()** interface.

Connection

Example

```

1  {
2    "baud": 500000,
3    "send_id": 2016,
4    "global_send_id": 2016,
5    "connection_type": {
6      "ISOTP": {
7        "blocksize": 8,
8        "st_min": 20
9      }
10    },
11    "server_type": "KWP2000",
12    "recv_id": 2024
13 }
```

properties

	Type	Description	Required
baud	Integer	The baud speed (bus speed) of the connection	Yes
send_id	Integer	The diagnostic tester ID	Yes
recv_id	Integer	The diagnostic receiver ID	Yes
global_send_id	Integer	The global tester present diagnostic ID	No
connection_type	Enum	The physical connection method to talk to the ECU. See below for a list of allowed data structures.	Yes
server_type	String	Denotes the diagnostic server type to use when talking to the ECU. Allowed values are "KWP2000" (Keyword protocol 2000), or "UDS" (Unified diagnostic services)	Yes

Connection type

Example (LIN)

```

1  "connection_type": {
2    "LIN": {
3      "max_segment_size": 254,
4      "wake_up_method": "FiveBaudInit"
5    }
6  }
```

Properties

	Type	Description	Required
max_segment_size	Integer	The maximum frame size allowed to be transmitted over K-Line	Yes
wake_up_method	String	Specifies the wake up method for K-Line. Allowed values are "FastInit" (Fast initialization wake up method) or "FiveBaudInit" (Five baud initialization wake up method)	Yes

Example (ISO-TP)

```

1 "connection_type": {
2   "ISOTP": {
3     "blocksize": 8,
4     "st_min": 20,
5     "ext_can_addr": false,
6     "ext_isotp_addr": false,
7   }
8 }
```

Properties

	Type	Description	Required
blocksize	Integer	The maximum number of CAN Frames allowed to be transmitted over ISO-TP before the ECU Must send another flow control message back to the tester. A value of 0 means Infinite block size (No flow control).	Yes
st_min	Integer	The minimum delay in milliseconds before sending consecutive CAN Frames to the ECU (Values between 0xF1-0xF9 represent 100-900 microseconds)	Yes
ext_can_addr	Boolean	Indicates if the CAN Frame's IDs are extended (29 bits - True), or standard (11 bits - False)	Yes
ext_isotp_addr	Boolean	indicates if the ISO-TP transport layer shall use extended ECU addressing or not.	Yes