FAKULTÄT FÜR INFORMATIK UND MATHEMATIK

HOCHSCHULE MÜNCHEN

Masterarbeit zur Erlangung des akademischen Grades Master of Science (M.Sc.)

An open source XCP based measurement and calibration system for automotive ECUs

Michael Marvin Wolf

FAKULTÄT FÜR INFORMATIK UND MATHEMATIK

HOCHSCHULE MÜNCHEN

Masterarbeit zur Erlangung des akademischen Grades Master of Science (M.Sc.)

An open source XCP based measurement and calibration system for automotive ECUs

Ein Open Source XCP basierendes Mess- und Kalibriersystem für Automotive ECUs

Autor:	Michael Marvin Wolf
Matrikelnummer:	01077012
Betreuer:	Prof. Dr.-Ing. Orehek
Abgabe der Ausarbeitung:	22.01.2018

HOCHSCHULE
FÜR ANGEWANDTE
WISSENSCHAFTEN
MÜNCHEN

Wolf Michael Marvin	München, 22.01.2018
(Familienname, Vorname)	(Ort, Datum)
07.08.1990	IG3 / WS 2017
(Geburtsdatum)	(Studiengruppe / WS/SS)

Erklärung

Hiermit erkläre ich, dass ich die Masterarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

(Unterschrift)

Abstract

This thesis gives an introduction to the universal measurement and calibration protocol XCP. The XCP protocol is used to perform online measurement and calibration of internal ECU variables. For this purpose, an open source XCP-master tool is developed: OpenXCP. It is further shown how a XCP driver can be implemented in an automotive ECU: XCP-slave.
In this project XCP on Ethernet is used as a transport protocol. OpenXCP introduces JSON as a data model file format, instead of the standard A2L (ASAP2) file format. Nevertheless, OpenXCP is still compatible with other XCP master software. Also details on the system and software architecture of master and slave is provided. Is explained how the algorithms for parsing the slaves ELF file with debug information (DWARF) works. OpenXCP also introduces an automatic method for extracting XCP measurement and calibration attributes directly from the ECU source code. An analyze shows the performance of OpenXCP in polling and synchronous data acquisition (DAQ) mode. The OpenXCP project is published on GitHub: `https://github.com/shreaker/OpenXCP`

Index terms— Universal Measurement and Calibration Protocol (XCP),
XCP on Ethernet, online measurement and calibration, open source, embedded system

Contents

Glossary

A2L ASCII file format for ASAM MCD-2 MC (ASAP2).

AML ASAM MCD-2 MC (ASAP2) MC metalanguage.

ASAP2 Defines the description for the data model for ECU Measurement and Calibration. Market name of ASAM MCD-2 MC.

CRC Cyclic redundancy check is used to detect errors in memory. It is used to calculate a checksum over the memories data to check integrity.

CSV Comma-separated-values. Stores values separated by a comma in an ASCII file.

CTO Command Transfer Object. Used for XCP commands and in for measurement in polling mode.

DAQ Data Acquisition.

DIE Debugging Information Entry. DWARF uses this as a data structure to represent variables, data types, subroutines, etc. A DIE is part of a tree structure and can have nested DIEs.

DTO Data Transfer Object.

DWARF Debugging data format.

ECU Electronic control unit.

ELF Executable and Linkable Format.

GCC GNU Compiler Collection which included the GNU C compiler.

Intel HEX A file format that stores binary data in an ASCII file. The HEX file is used for flashing ECUs.

Master A software tool to measure and calibrate one or more slaves using the XCP protocol.

ODT Object Descriptor Table.

PID Packet Identifier.

Slave An ECU with an XCP driver. The master measures and calibrates internal ECU variables of the slave.

STIM Synchronous Data Stimulation.

XCP Universal Measurement and Calibration Protocol used to measure and calibrate ECU software and algorithms.

1. Introduction

1.1. Background and motivation

The development of software for embedded systems is a challenging task. Many steps from requirements engineering, to implementation and calibrating, to deploying the software must be taken. Implementation and configuration of an embedded system is often more difficult compared to software development for general purpose computers. This is because an ECU is very limited in its resources. Also an ECU is embedded in a surrounding technical system, like a vehicle. Such a technical system is complex and consists of a number of ECUs working together. Also the system has to interact with the physical world. Therefore, safety and security standards apply.
For developing embedded software, especially complex algorithms, it is essential to evaluate and then configure the software. This is best done under the real environment of the physical system. Aggravating this situation, in the automotive sector the technical system, a vehicle, is a moving object. Well-known processes for monitoring and debugging are not sufficient here. The software on the ECU must run in production mode and not with a debug information overhead. This is because often the available memory space on the ECU is not enough for running a non-optimized code. Also the algorithms behavior should be tested and configured under the impact of non-artificial events.
In the automotive sector the firmware development and the calibration of the system is divided in separate responsibilities. A software developer designs and implements an algorithm and a calibration engineer configures the parameters of the system. The calibrator is provided with a set of tools for measuring and calibrating the highly technical parameters. With a background in engineering the calibrator is more suited for this task. Another benefit of this approach is that the source code must not shared. Measurement and modification of the algorithms parameter are done by a measurement

and calibration protocol. This makes the process of altering parameters easier and time saving. Source code modification, compiling, linking and flashing the ECU is not necessary anymore. Also the need of sharing source code including intellectual property is eliminated. The measuring and calibration tool only needs the binary image for the ECU and a special description file called A2L.
A solution for the described problem is XCP. XCP stands for 'Universal Measurement and Calibration Protocol'. It is standardized by ASAM e.V. (Association for Standardization of Automation and Measuring Systems). The main use case for XCP is to measure and calibrate internal ECU parameters during runtime. Without the use of XCP an ECU algorithm can be considered as a black box. The calibrator can only monitor the input and output signals of the ECU, but can not evaluate the internal behavior of the algorithm. This is not sufficient for understanding and optimizing an algorithm.
XCP enables the calibrator to see the value of a parameter by accessing the variables memory address inside the ECU. To modify a parameter, again XCP accesses the variables memory address, but this time with write access instead of read access. By this approach an algorithm can be improved iteratively. This is done by measuring parameters, evaluating, modifying parameters and starting all over again until the algorithm has reached a production ready state. XCP allows to measure parameters synchronous to an internal event of the ECU. This allows to set different parameters in correlation to each other. Also timestamps can be retrieved, which further improves the information value of the measurement.
XCP is a protocol used by an so called XCP master tool to measure and calibrate a slave. The master tool is a software running on a personal computer and the slave is an ECU including a XCP driver. A slave is not necessarily always an ECU. Every system, such as a simulation software, can be used with XCP. This is for example useful for hardware-in-the-loop development, or developing algorithms with MATLAB/Simulink [The18].

XCP is based on CCP (CAN Calibration Protocol). XCP is the successor of CCP and among other major improvements, XCP supports a broad number of transport protocols. These transport protocols are interchangeable. XCP is an application layer protocol, that defines an interface for the underlying transport layer protocols.
On the market solutions for measurement and calibration tools already exist. They either user proprietary protocols and tool chains or the XCP protocol. They both have

in common that they are expensive and knowledge about using the tools is mostly only known by experts of the field. Companies provide free-to-use demonstration applications of an XCP master tool and a XCP slave driver, but with serious limitations. For this reason the goal of this thesis is to understand the measurement and calibration processes used by XCP and to develop an open-source XCP master tool. By the time of writing this is the only free and open-source measurement and calibration tool based on XCP. The tool is called OpenXCP and will support development of embedded firmware. Especially for student projects at the Munich University of Applied Sciences. Starting point for this project was a previous project concerning a model car. The car is controlled by an Infineon Aurix TriCore ECU equipped with a line scan camera. The objective was to develop a firmware and an algorithm so that the car follows a black line. The line scan camera is able to distinguish between dark (black) and bright light. The relating path-algorithm was responsible for holding the vehicle on track, while handling changing light conditions. Fine-tuning and adjusting the algorithms parameter proved to be very tricky. No adequate tool was available for evaluating the algorithms behavior, while the car was trying to follow the black line. The measurement of parameters could not be done in during a debug session. Simple because the ECU must control the cars motors in real-time, by generating a PWM signal, and react to interrupts from the line scan camera. Setting a breakpoint in debug mode would lead to malfunction, because the execution of the ECU software is stopped while the motors keep running. Also the measurement progression of a parameter by time or by an event could not be logged to a file. Adjusting an algorithms parameter was done in a time consuming manner: The value of the parameter had to be changed by editing the source code, compiling the code and then flashing the HEX image to the ECU. Offline-calibration concepts could have been used, but they have their own drawbacks. An offline-calibration is for example, editing a parameters value in the ECUs HEX file and flashing the HEX file on the ECU again. Source code modification is therefore not necessary.
With XCP it is possible to overcome this issues and speed up development and calibration. OpenXCP will be able to measure internal ECU parameters in RAM during runtime and save them to a record file. The measurement will be time or event based and the measurement parameters are in relation to each other. Online calibration of parameters will be possible, without modifying the source code or the HEX file.

1.2. Structure of thesis

2. Goal of OpenXCP Use cases and requirements are defined for the XCP master tool and the slaves XCP driver as part of the requirement engineering.

3. Fundamentals A technical look inside the XCP measurement and calibration process.

4. Architecture Description about the system and software architecture of the OpenXCP master tool and the slaves XCP driver.

5. Development process and software implementation Explanation about relevant implementation decisions, algorithms and insights on the development process.

6. Result Conclusion about the OpenXCP project. This includes an analyze of the measuring and calibration process with OpenXCP. Also problems encountered during development and future work are mentioned.

2. Goal of OpenXCP

This chapter describes use cases and requirements for OpenXCP master and slave. Use cases show the software from a user-oriented perspective which takes an external actor (user) into account. Whereas, the functional requirement view on the project is a approach to describe the software in more detail for developers.

2.1. Use cases

2.1.1. Master

This section briefly describes the use cases for the OpenXCP master. The actor used for the use cases is an engineer with a background in measuring and calibration of embedded systems. Figure 2.1 on page 9 shows the UML use case diagram.

Use case:	**Measure parameter**
Description:	Measure value of variable from slave
Preconditions:	1. Connection between master and slave established
Basic flow:	1. Parse ELF/DWARF: Memory address/size of variable is determined 2. Parse XCP comment: XCP settings for variable are determined 3. Connection is established between master and slave 4. Variable value is received by polling or DAQ 5. Variable value is displayed and saved including a time-stamp

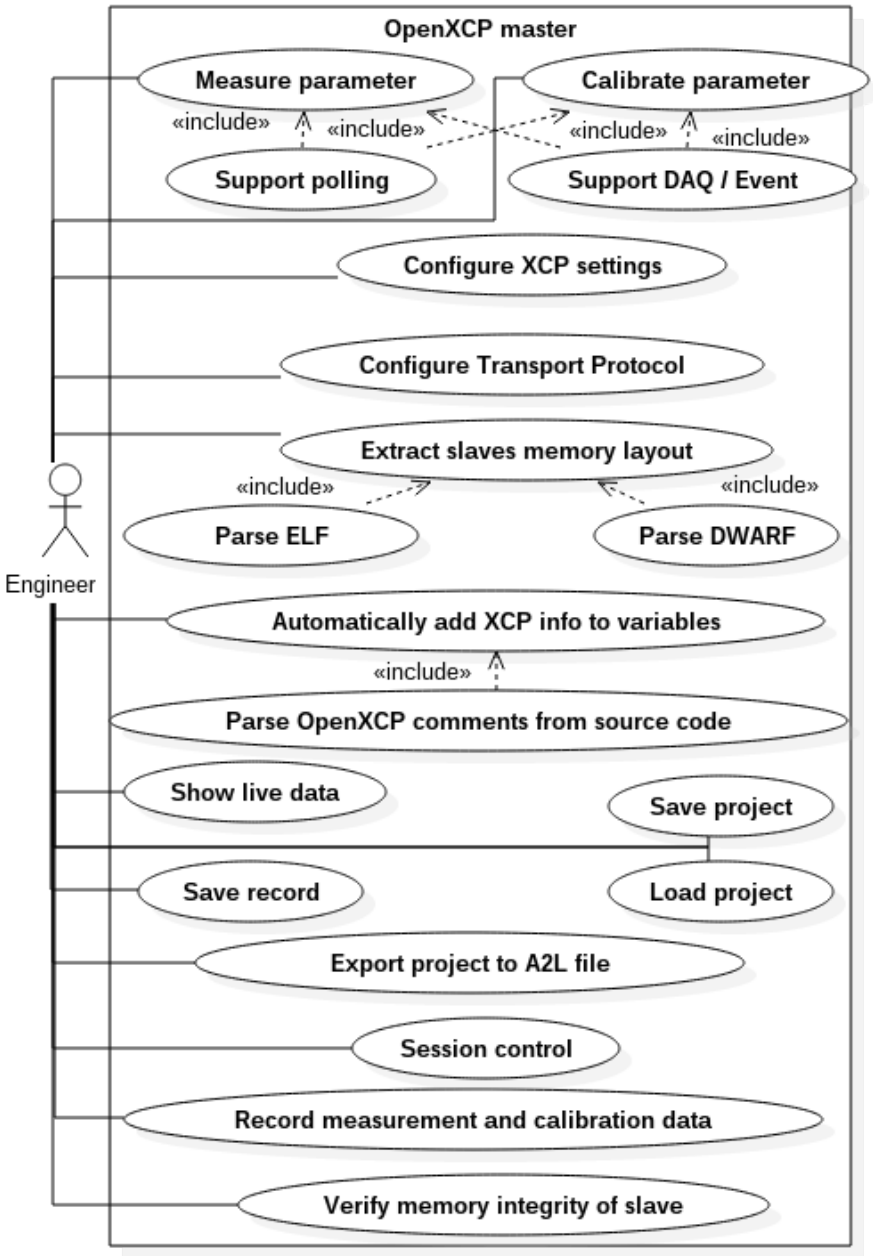


Figure 2.1.: Use cases master

Use case:	**Calibrate parameter**
Description:	Modify value of variable from slave

Preconditions:

1. Connection between master and slave established

Basic flow:

1. Parse ELF/DWARF: Memory address/size of variable is determined
2. Parse XCP comment: XCP settings for variable are determined
3. Connection is established between master and slave
4. New variable value is entered by engineer
5. Variable value is updated in the slaves memory
6. Variable value is received by polling or DAQ for verification
7. Variable value is displayed and saved including a time-stamp

Use case:	**Configure XCP settings**
Description:	Set XCP configuration to match slaves requirements

Preconditions:

1. Slaves XCP configuration must be documented

Basic flow:

1. Adjust basic XCP settings: version/XCP packet size/Endianness ...

Use case:	**Configure transport protocol**
Description:	Set underlying transport protocol for XCP

Preconditions:

1. Slaves transport protocol setup/configuration must be documented

Basic flow:

1. Set appropriate link layer protocol (Ethernet, USB)
2. Ethernet: Configure IP connection settings: ip, port
3. Ethernet: Choose and configure transport layer (UDP, TCP)

Use case:	**Extracts slaves memory layout**
Description:	Parse ELF file from slave and analyze DWARF information

Preconditions:

1. ELF file must match the slaves current memory

Basic flow:

1. Parse ELF file for name, address and size of variables
2. Parse DWARF tree for additional information about variables
3. Show result of parsing to the user

Use case:	**Automatically add XCP info. to variables**
Description:	Automatically add XCP information to each variable by parsing the slaves source code for OpenXCP comments

Preconditions:

1. Special OpenXCP comment must be added to slaves source code above the declaration of a variable
2. Slaves source code must be accessible with read privilege
3. DWARF information must be parsed (source code file path of each variable)

Basic flow:

1. Parse ELF file for name, address and size of variables
2. Parse DWARF tree for source code file path of each variable
3. Parse source code for OpenXCP comments and match them with the correct variable

Use case:	**Show live data**
Description:	Show the user the value of the monitored variables

Preconditions:

1. Variables must be selected for monitoring (measurement or calibration)
2. Connection between master and slave established

Basic flow:

1. Select variables for monitoring (measurement or calibration)
2. Connect to slave
3. Start record XCP session (polling or DAQ)
4. Receive and show current value of monitored variable

Use case:	**Record measurement and calibration data**
Description:	Save the received value of each variable in memory with a time-stamp

Preconditions:

1. Connection between master and slave established

Basic flow:

1. Select variables for monitoring (measurement or calibration)
2. Connect to slave
3. Start record XCP session (polling or DAQ)
4. Receive and save current value of monitored variable including a time-stamp

Use case:	**Save record**
Description:	Write the received values of all variables to a CSV file

Preconditions:

1. Value and time-stamp pair must be saved in memory for all monitored variables
2. Stop record of data

Basic flow:

1. Record session is running and recording value and time-stamp pairs for each monitored variable
2. Recording session is stopped
3. Value and time-stamp pairs are written from memory to a CSV file
4. Evaluate or plot data in CSV file with a third party tool (MATLAB, EXCEL, CANape)

Use case:	**Session control**
Description:	Connect/disconnect to slave, start and stop measurement and calibration session

Preconditions:

1. XCP settings must match slaves requirements
2. Transport protocol setting must be configured correctly

Basic flow:

1. Connect to slave
2. Start measurement and calibration session
3. Stop measurement and calibration session
4. Disconnect from slave

Use case:	**Save project**
Description:	Save XCP and transport protocol setting. Save selected variables from parse process

Preconditions:

1. Project file name and path selected

Basic flow:

1. Save project settings, including ELF file hash value, to new or existing JSON file

Use case:	**Load project**
Description:	Load XCP and transport protocol setting. Load selected variables from prior parse process

Preconditions:

1. Project file must be saved

Basic flow:

1. Load project JSON file and update the state and data of OpenXCP
2. Compare if ELF file has been modified (hash value in project file compared to hash of ELF file on the computer)

Use case:	**Export project to A2L file**
Description:	Export/save XCP and transport protocol setting and selected variables from parse process in ASAP2 format to an A2L file

Preconditions:

1. A2L file name and path selected

Basic flow:

1. Save project settings to an A2L file in ASAP2 format
2. Load A2L file with third party XCP-master tool (e.g. CANape)

Use case:	**Verify memory integrity of slave**
Description:	Verify if the memory layout in the ELF file matches the current slaves memory.

Basic flow:

1. Calculate the hash value from the salves HEX file
2. Send a request to the slave to calculate the hash value of its memory
3. Compare the hash value

2.1.2. Slave

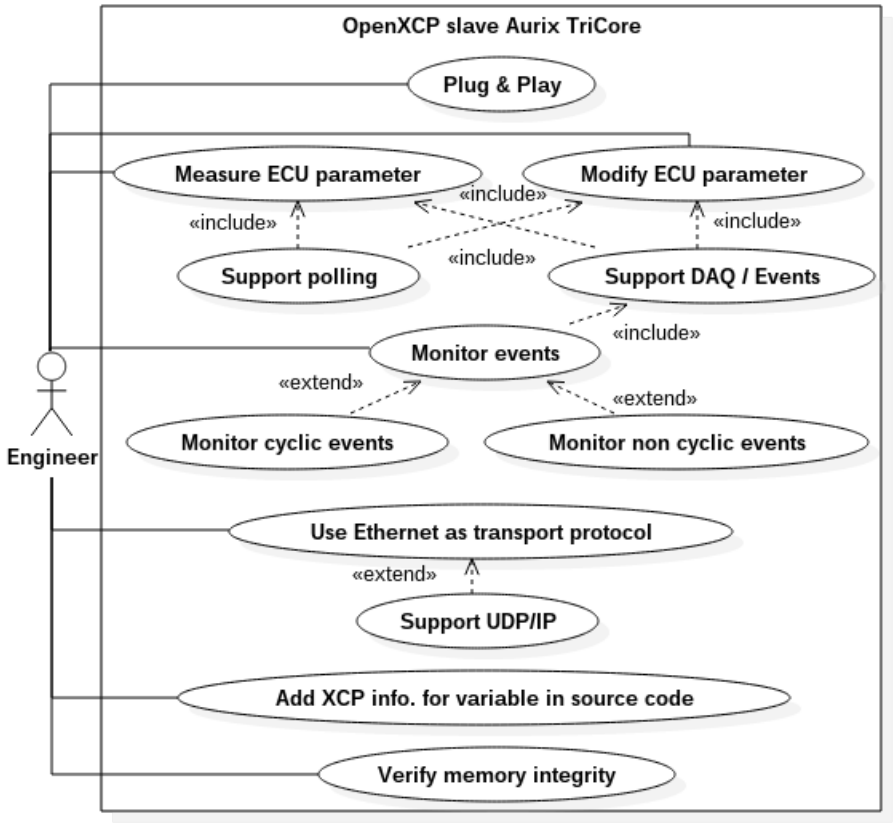


Figure 2.2.: Use cases slave

This section briefly describes the use cases for the OpenXCP slave implementation on an Aurix TriCore ECU with Ethernet. The actor used for the use cases is an engineer with a background in measuring and calibration of embedded systems. Figure 2.2 shows the UML use case diagram.

Use case:	**Plug and Play**
Description:	Easy configuration of XCP driver to measure and calibrate parameters with a XCP master

Basic flow:

1. Adjust XCP driver parameters by altering the preprocessor directives in the ECU source code. Parameters must fit the project and ECU hardware requirements.
2. The master is configured with matching XCP settings
3. Master establishes a connection with the slave
4. Master requests the slaves XCP settings and checks if they match the masters settings
5. Master starts the measurement and calibration session

Use case:	**Monitor ECU parameter**
Description:	Read the value of a variable during runtime from memory and send the data to the master.

Preconditions:

1. Master has the correct information about the variables memory address. Achieved by parsing the ELF file which was flashed onto the slave.
2. Slave and master are configured correctly and have established a connection.

Basic flow:

1. Master requests the value from the memory address of the variable by polling or DAQ XCP-commands.
2. Slave reads the value from the memory address and sends the value back to the master.

Use case:	**Modify ECU parameter**
Description:	Alter the value of a variable in memory during runtime.

Preconditions:

1. Master has the correct information about the variables memory address. Achieved by parsing the ELF file which was flashed onto the slave.
2. Slave and master are configured correctly and have established a connection.

Basic flow:

1. Master sends the memory address of the variable and the new value to the slave.
2. Slave updates the value of the variable at the given memory address.
3. Master requests the value from the memory address by polling to check if the update was executed correctly.
4. Slave reads the value of the memory address and sends the value to the master.

Use case:	**Monitor events**
Description:	Notify the master about an internal event in the ECU. An event can be cyclic (timer) or non cyclic.

Preconditions:

1. Master has the correct information about the variables memory address.
2. Slave and master are configured correctly and have established a connection.

Basic flow:

1. Session start: Master sends DAQ list including variable and event pairs to the slave
2. Continuously during a session: Slave sends the value of a variable, if the associated event is triggered.

Use case:	**Use Ethernet as transport protocol**
Description:	Master and slave exchange XCP commands over the Ethernet (UDP/IP) transport layer.

Preconditions:

1. Master and slave are physically connected via Ethernet cable
2. IP address and port of master and slave are configured correctly.

Basic flow:

1. XCP command payload is embedded in an UDP packet and exchanged between master and slave

Use case:	**Add XCP information for variable in source code**
Description:	Additional XCP information for a variable can be included in the ECUs source code.

Preconditions:

1. Slaves source code must be accessible with read and write privileges

Basic flow:

1. XCP information is added above the declaration of a variable as special OpenXCP comment.
2. Master parses the source code for these comments and automatically adds the information to the matching variable.

Use case:	**Verify memory integrity**
Description:	Calculate a hash value over the memory using a hash algorithm

Basic flow:

1. Master requests to calculate a hash value over a certain part of the memory
2. Slave calculates hash value with a specified hash algorithm (CRC, ADD) and sends the result to the master

2.2. Functional requirements master

A functional requirement describes what function the XCP master tool must provide to solve an identified task [Som07].

Measure parameter The software must be able to receive and process the current value of a parameter from the slave during runtime. A parameter is a basic variable, an array or struct member, which is selected for measurement. The received data must be interpreted correctly. At first, all signed and unsigned integer data types must be supported. Next, floating data types must be supported.
Polling- and event-driven-mode (DAQ) must be provided for measuring a parameter. For the DAQ-mode the user must be able to assign a cyclic or non-cyclic event to a parameter. The event in the master software must match the configured event in the slave. For polling mode a cyclic timer must be provided to request the parameters value.

Calibrate parameter It must be possible to modify a parameter while the runtime of the slave. As described in the requirement 'Measure parameter', basic variables, array-, struct-members and the mentioned data types must be supported for calibration. The user must be able to enter a new value for the parameter, which is selected for calibration. This value must be in the allowed range. Maximum and minimum value

as well as step size is provided by the calibration engineer. This is done by adding OpenXCP comments to a parameter in the ECU source code or by manually entering the information in the XCP master tool. After entering the new value in the master the value is send to the slave. The slave updates the value of the parameter. The master then must request the value of the parameter and check if update worked. The request must be done in polling mode. A parameter selected for calibration is also monitored, like a measurement parameter, by polling- or event-driven-mode (DAQ).

Record measurement and calibration value The software must be able to save the values of parameter during an measurement and calibration session. For every monitored parameter a pair of current value and time-stamp must be saved. This must be done every time the slaves sends a value for a parameter to the master. The user must be able to save the recorded data to a CSV file. This allows the user to export the data to another software, like EXCEL or MATLAB [The18], to further work on the data set.

GUI for user interaction A graphical user interface must enable the user to control all necessary features of the XCP master tool:

- Project management (save/load project file)
- Export to A2L file
- Parser management
- Show all information produced by the parsers
- XCP configuration
- Transport protocol configuration
- Event configuration
- Configure variables for measurement and calibration
- XCP session control
- Show current value of monitored parameters
- Input for new calibration value

Extract slaves memory layout The software must be capable to parse ELF files produced by a GCC compiler. For this reason an open source library which is able to extract data from an ELF file must be chosen. The parser reads the ELF file image which was flashed in the slaves memory. Therefore, the ELF file represents the memory layout of the slave, which includes the memory address of all global variables. These variables can than later be monitored or modified by the master. For this reason the ELF parser must extract the symbol name, memory address and the variables size in bytes from the ELF file. The DWARF parser reads the debug information. The DWARF parser walks through the DIE tree to find additional information: abstract data types (array, struct), basic data type, source file path and file line number of all global variables. The DWARF parser must be able to interpret the DIE tree so that the software knows the composition of a struct down to basic data type members. This is necessary because XCP only works on basic data type variables and not on composite data types.

Parse OpenXCP comments from source code Each parameter from the ECU needs additional attributes to be compatible with XCP protocol. The ASAM standard for ASAP2/A2l describes the mandatory and additional attributes for measurement and calibration parameters. For measurement and calibration of parameters this information is needed in addition to the data obtained from the ELF parser. The master software must provide a parser for automatically reading the attributes for each variable from the ECU source code files. The attributes differ between measurement and calibration parameters. Therefore, the variable must be marked for either of the both. For this project following attributes are selected as mandatory:
Measurement: is discrete, max. refresh rate, conversion function, is write able
Characteristic: step size
In common: long identifier, lower- and upper limit, physical unit
Further attributes may be added in a next version of the software. All OpenXCP attributes must match the attributes defined by the ASAM standard for ASAP2/A2l.

Save project settings to a file It must be possible to save the project settings to a file and continue work after a restart of the software. The project save file must be written to a file in a standard format (XML, JSON, A2L). The file must be compatible with standard third party XCP master tools (e.g CANape).

Verify memory integrity of slave To avoid reading or writing to an incorrect memory address, the master software must verify if the memory layout obtained by parsing the ELF file match the actual memory of the slave. Therefore, the master must calculate a hash value over the slaves HEX file. The HEX file contains the actual data flashed onto the slave, without any debug information. The master than must command the slave to calculate the hash value over its memory. The same hash algorithm must be used by master and slave (CRC, ADD). The hash values are than compared by the master to determine the integrity. The flash memory must be compared as well as the RAM memory. The later must be verified to avoid initialization errors of variables.
The software must also verify if the ELF file has been modified after loading an previously saved project. If a project is saved a hash value of the ELF must be saved in the project file. This saved hash value must be compared with a new hash value calculated for the ELF file. If they match, the ELF file was not modified and the user continue work without parsing the ELF file again.

2.3. Non-functional requirements master

Non-functional requirements specify attributes that define how the software should be. This includes quality and measurable criteria [Som07].

Open source The software shall be released under an open source license. It would be an alternative to commercial XCP master tools. Also other developers can tweak or help to improve the software. This is useful for universities, which want to use a free XCP master tool for measuring and calibrating embedded software. It should be assured that the used third party libraries and frameworks used for this XCP master meet the open source license requirements.

Multi operating system support The tool shall be development in a multi platform programming language/framework. First, the software shall be released for Microsoft Windows 7/10.

User experience and software quality The GUI shall be responsive while exchanging XCP messages with the slave. A multi-threaded approach is therefore needed to keep the user interface reactive while preforming background task. The user should be informed about the current status of the software. Helpful messages can be used to inform the user about all relevant tasks. As parsing the ELF file and the source files can take some time, a progress bar shall inform the user about the progress. The software should also be intuitive to use for an experienced calibration engineer or a student with a engineering background.
The software shall be developed in such a way that is maintainable and easy to add new features for other developers. This is especially worth mentioning as the software will be released as open source and should have low boundaries for new developers to join the project.

High performance measuring and calibration system The software shall be able to handle a great number of parameters simultaneously. This includes parameters for measurement as well as for calibration. The system should be capable of sending polling requests with a minimum of 10 ms. Timing shall not differ more than 1ms from the given request time interval. While polling the software should still be able to receive XCP packets send by the slave asynchronously (DAQ).

Error handling General error handling shall be implemented. Especially for the 'XCP over UDP driver' an appropriate error hanlding should be developed. This includes:

- Handle XCP packet loss
- Start timeout counter for XCP packets
- Keep order of received and send XCP packets
- React on error messages send by the slave

2.4. Functional requirements slave

Support XCP protocol (mandatory) A driver for the XCP protocol must be implemented to support all mandatory XCP commands and tasks. This includes Command Transfer Objects (CTO) and Data Transfer Objects (DTO). Following CTOs must be supported: Command Packet (CMD), Command Response Packet (RES), Error (ERR). Following DTO must be supported: DAQ (Data AcQuisition). Support for non-mandatory XCP commands must be added as the project makes use of an additional XCP feature. The driver must provide an API for underlying transport protocols, such as Ethernet, USB, and so on. The XCP driver must be able to access the ECU memory to execute measurement and calibration tasks.

Support Ethernet UDP/IP as transport protocol for XCP For the Infineon Aurix TriCore ECU Ethernet must be used as a transport protocol for XCP. UDP is chosen, because of its low latency, low resource demand and its high packet throughput. While measuring a parameter it is important to receive the current value in time. If one packet is lost, than this not a problem, because the next request will get a new value for the parameter shortly after. A XCP payload is embedded in an UDP packet. A UDP packet can carry several XCP packets to reduce UDP packet overhead. The transport layer driver must interact with the XCP driver via the XCP drivers transport layer API. The interface between the drivers allows to easily replace the transport layer of XCP with another technology, for example USB.

Support XCP polling requests and DAQ/Events The XCP driver and the transport layer driver must support polling requests from the master. This means the master sends a request as a CMD packet and the slave replies with RES or ERR packet. The slave must support the configuration of DAQ lists. The slave then asynchronously sends packets to the master if an internal event occurs.

Support memory integrity test The slave must support an ADD or CRC algorithm to calculate a hash value for a given memory region. The computation can be split up. Therefore the processor is not blocked for a long time while calculating the hash value

in a background task. The slave keeps sending intermediate results to the master, until the hash value for the complete memory region is calculated.

2.5. Non-functional requirements slave

Low resource requirements for XCP driver The XCP driver and the transport layer driver should use low RAM and CPU resource. The use of RAM memory shall be configurable. The benefits of adjustable resource consumption is, that the same XCP driver can be used on various ECUs.

Simple integration of XCP driver to an existing ECU firmware It shall be easy for a developer to integrate the XCP driver to his existing ECU firmware. The existing firmware shall be enhanced with the measurement and calibration features of XCP, without the need of modifying existing business code.

3. Fundamentals

3.1. Basics

The complexity of automotive ECU software is constantly increasing. Not only the individual task of a single ECU gets more complicated, also the overall system has increasing demands. A highly automated vehicle is controlled by numerous ECUs, with different safety levels (ASIL). The more software and ECUs, that get added to a vehicle, the more effort it takes to test, calibrate and integrate the components. Safety critical components must also meet the requirements of standards like ISO 26262 [Nor11]. This altogether results in the demand of reliable, effective and simple to use measurement and calibration tools. [CLG11, p.1]
For the purpose of online measurement and calibration of ECU software, ASAM e.V. standardized the XCP protocol. ASAM stands for Automotive Association for Standardization of Automation and Measuring Systems. Participants are mostly companies and organizations from the automotive sector. XCP is a abbreviation for 'Universal Measurement and Calibration Protocol'. In 2017 version 1.4.0 was released [ASA17a]. XCP is the successor of CCP (CAN Calibration Protocol). It improves CCP in many ways, but most notable is that XCP is divided in an application layer and a transport layer. XCP supports a wide variety of underlying transport protocols, such as Ethernet, USB and CAN. [PZ16, p. 7]

The main goals of XCP are:

- To use minimal ECU resources
- Simple implementation of XCP-driver for ECUs
- To provide an efficient communication protocol with minimal overhead
- Scalable, to support a wide variety of hardware and project requirements
- Easy and quick to configure XCP settings for slave and master

[PZ16, p.12]

3.2. Use cases

This section will briefly introduce straightforward and advanced use cases of XCP. Also benefits of XCP compared to other testing and calibrating technologies will be highlighted.

Distributed systems It is challenging to calibrate or test software functions, that are part of a distributed system. ECUs in a vehicle interact with each other over an on-board network. A function of an ECU can cooperate with a function from another ECU. Also a function by itself can be distributed over several ECUs. This makes it hard to find an error, when so many software components are involved. Traditional debugging methods are very limited in their support when it comes to errors that occur sporadically or only in the test vehicle. Rest bus simulations help to verify software functions at a early development stage, but fail if the ECU is deployed in a vehicle. To find an error in a distributed system, it is necessary to measure the value of parameters from different ECUs simultaneously. The measurement is executed at a given trigger point for reference purpose. This makes the measurement of the parameters correlate to each other. The correlation of parameters and timestamps help to construct an record history and to find errors that occur across ECU boundaries [MA17, p.1f.]. This is important, because distributed systems lack a global state and a global clock [Sch94, p.4].

When it comes to calibration of distributed systems it is essential to see the effect of a parameter modification in all involved software modules simultaneously. This allows to determine the consequence of the calibration and to analyze the system behavior. [MA17, p.1 f.] [He 09]
Also physical limitations reduce the use of laboratory debugging methods:

- No space to access ECU or to install debug tool in the vehicle.
- Debuggers for different ECUs do not output standardized data.
- The ECU has no connection port left for debugging.

[MA17, p.1]
XCP helps to test and monitor distributed systems. XCP is a single master, multi slave protocol. Therefore, a master tool can interact with several slaves at the same time. The performance of a system with a master and two slaves was tested by [He 09]. They used the synchronous data transfer mode of XCP to exchange data between master and slaves (DAQ). DAQ uses internal ECU events to trigger a measurement and to send the parameter data to the master. XCP allows to add timestamps to every measurement. As known, XCP is standardized so that the measurement of different hardware produces a standardized output. Also the physical limitations in test beds are addressed: XCP allows the use different transport protocols. This means that no extra hardware, like a JTAG connector, must be attached to an ECU. The given transport communication interface can be reused. [PZ16, p.14] Overall XCP is very well suited for testing and measuring distributed systems [He 09, p.7].

AUTOSAR combined with XCP XCP can be combined with the Automotive Open System Architecture (AUTOSAR). AUTOSAR is a standard, that defines a modular architecture for ECU software. The goal is to develop software, which is reusable, maintainable and scalable for different ECUs. This is done by a layered architecture. It separates functional software from hardware specific code. [CLG11, p.2]
Several compliant implementations of the AUTOSAR standard exist. For example, MICROSAR and EB tresos AutoCore both support AUTOSAR version 4.x and 3.x [Gmb17], [Ele17]. AUTOSAR 4.x includes a monitoring and debugging module called AMD. Figure 3.1 on page 30 shows how the modules of AMD are connected to a XCP Master tool. In this case it is the MICROSAR AMD implementation, which pro-

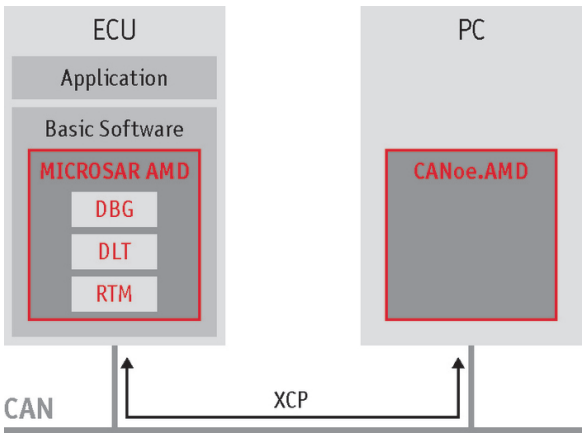


Figure 3.1.: AUTOSAR monitoring and debugging [Onl14]

vides access to internal AUTOSAR-ECU parameters. The first module is the debug interface (DBG) that is used to access the status of a basic software (BSW) module. Secondly, the diagnostic log and trace (DLT) module is responsible of tracing internal ECU activity and logging them to a text file on a computer. The last module is the runtime measurement module. It can be used for profiling the application and the basic software. For example measuring the CPU load or the runtime of an interrupt routine or a task. [Onl14], [Gmb17, p.61 ff.] AUTOSAR specifies a dedicated communication protocol to communicate with a computer tool. The AUTOSAR protocol is not compatible with XCP. Like XCP, which uses an A2L file for calibration and measurement configuration, a description for debugging is generated. This is based on the BSW modules descriptions. [MA17, p.29 f.]
However, for transferring the data from AUTOSAR to a PC tool, XCP can also be used as shown in the figure above. MICROSAR and several OEMs continue to use the XCP standard for measurement and calibration of AUTOSAR ECU software. The benefit of XCP compared to the AUTOSAR protocol is that existing XCP based tools can be continued to be used. Also the work-flow of XCP is well-known. For the AUTOSAR protocol an additional BSW module has to be configured and integrated. This effort can be saved by continuing to use XCP. AUTOSAR 3.x does not officially specify a remote debug mechanism. However, MICROSAR supports XCP for AUTOSAR version 3.x. [Sch14]. Care must also be taken, that not every AUTOSAR 4.x hardware platform supports XCP yet cite[p.2]TestsAutosar. The latest AUTOSAR 4.3.1 XCP module

specification is defined by this document: [AUT17].
The A2L description file for XCP can be generated from the AUTOSAR ECU configuration file (ECUC), with tools such as DaVinci from Vector. This makes it easier to integrate the XCP work-flow and requirements into AUTOSAR [Gmb17, p.63].
The following papers show how to use AUTOSAR and XCP together for testing automotive ECUs: [CLG11] and [MA17]. A more general perspective on testing automotive software is discussed by [D17]. A AUTOSAR conform test-bench is developed for static and dynamic tests of AUTOSAR modules.

Simulation A common method to develop and test an embedded software for ECUs is to use simulation methods. The ECU software in the simulation can be measured and calibrated by XCP before deploying it to the target hardware. This reduces time to market and costs. The target ECU hardware can be simulated as well as the plant. The term plant describes the physical system that is controlled by the ECU software. For example a vehicle engine. Hardware-in-the-loop (HIL), software-in-the-loop (SIL) and model-in-the-loop (MIL) simulation methods are used for this. Each simulation has a specific use case and benefit during development. Hardware-in-the-loop simulates the plant, while the software under test runs on the target ECU. This means, that all related parts of the vehicle are simulated, including input and output communication. The plant model must be capable to run in real-time, because the ECU software also runs in real-time. Hardware-in-the-loop test-beds can be low-cost to very cost intensive, depending on the requirements of the hardware, which must be simulated. A cheaper method is software-in-the-loop simulation. Here, both the plant and the ECU software under test are simulated by a PC software. In this approach the actual source code of the ECU software is tested. Compared to model-in-the-loop, which tests an algorithm in a model approach without the actual target source code.
[PZ16, p.92 ff.], [MPC08, p.1 f.]
All three simulation methods have in common that they can be used in combination with XCP for calibrating and measuring the algorithms and ECU software.
In case of hardware-in-the-loop the testing and modifying of the software is straightforward. The XCP master tool is connected to the ECU slave (including a XCP driver) as usual. Advanced test setups simulate the plant model as a DLL in the XCP master tool (e.g. CANape) or on a real-time server (e.g. CANoe). This makes it possible to

simultaneously calibrate and monitor the ECU software under test and the behavior of the plant.[PZ16, p.95]
Software-in-the-loop and model-in-the-loop use MATLAB/Simulink [The18] to simulate the ECU software and the plant. Simulink can generate C code, a DLL and a associated A2L file of the software under test. A XCP master tool can than be used to test the algorithms included in the DLL. For this the DLL must include a XCP driver to being able to establish a XCP session.[PZ16, p.93 f.]
[MPC08] discusses how to to use MATLAB S-Functions with the XCP master tool CANape for testing a ECU software. The entire software is embedded in a S-Function. The parameters of the software are accessed by XCP over Ethernet. The software-in-the-loop simulation can not executed in real time, due to running on a standard PC. Therefore, the XCP master tool and Simulink must be synchronized by using DAQ events and by transmitting timestamps.

High performance measurement The increasing complexity of ECU software and the high performance embedded hardware lead to a demand for high performance measurement and calibration solutions. Todays engine control units use PWM signals with a frequency up to 100 kHz. XCP on CAN or FlexRay can only measure internal ECU parameters with a maximum of 1 kHz. This is no sufficient. XCP over Ethernet increases the maximum bandwidth and the data rate. Therefore, it is possible to monitor a larger number of signals at a higher sampling rate. Nevertheless, additional measurement hardware is needed to even increase the measurement rate and to keep the CPU load of the ECU, under test, low. [Kle12]
XCP version 1.3 introduces this measurement hardware module to the standard. Before, the standard considered that the XCP master tool is always directly connected to the slave. This has changed. Now, a so called 'Plug-on-device' (POD) can be connected to the ECUs debug interface and to the measurement hardware module. The measurement hardware module is connected to the XCP master over XCP on Ethernet. The hardware unit acts as a protocol converter between the ECUs debug interface and XCP on Ethernet. [Kle12]
Two different ECU interfaces are available: a debug- and a data trace-interface. Normally these interfaces are used by debuggers, for example JTAG. Depending on the interface the measurement data rate, minimum frequency and ECU CPU load dif-

fers. For high performance measuring the DAQ mode of XCP is used. If available on the ECU a data trace interface is more powerful compared to a debug interface. For example the Aurora data trace interface supports a DAQ data rate of 30 MB/s, 5 GHz frequency and a minimal cycle time of 15 µs, if combined with a hardware measurement module. [A15]
Two different measurement methods can be used: RAM copy or data trace. The first measurement method is used for debug interfaces, which is referred as 'Online Data Acquisition' (OLDA). Here the internal measurement signals are copied from RAM and send to the master. This costs CPU load for the slave. The second method can only be used with data trace interfaces. The hardware measurement module keeps a exact copy of the ECUs RAM in memory. A software trigger in the ECU triggers the measurement module, which saves changes in the buffered memory. The mirrored memory of the measurement module is read and send to the XCP master. This results in almost no CPU load on the ECU and high measurement performance. [A15], [Kle12]

3.3. Protocol layer

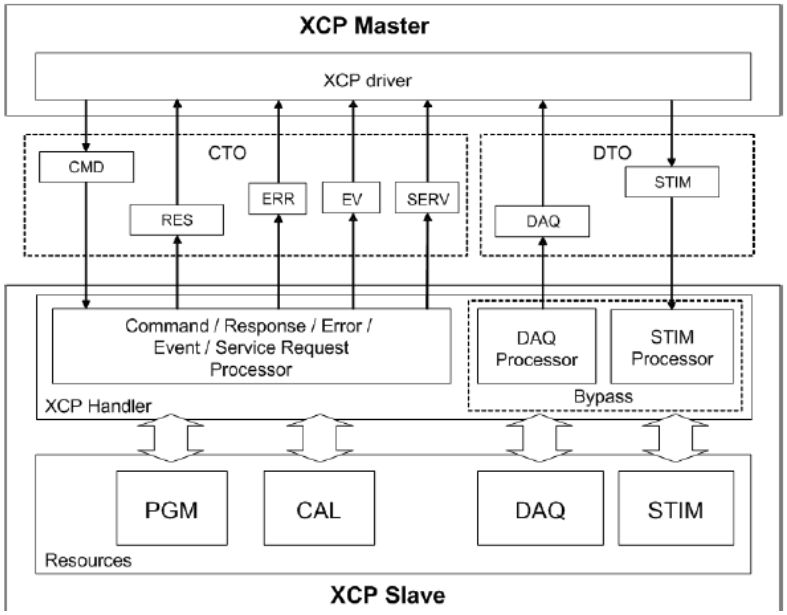


Figure 3.2.: Master slave communication overview [ASA15b, p.83]

Overview XCP packets are independent of the underlying transport protocol. The XCP packets are exchanged between master and slave in a packet-orientated way. The master initiates the communication and the slave response to the message. The figure 3.3 on page 34 shows shows a XCP message frame. The XCP message frame includes a header, a packet and a tail. The XCP message frame is embedded in the packet of the transport protocol. The structure and the content of a XCP header and a XCP tail depend on the transport protocol. The XCP packet is generic and independent from the transport protocol. The packet contains a mandatory identification field, an optional counter field, an optional time-stamp field and an optional data field. Every packet is identified with a packet identifier (PID). The master and the slave can therefore unambiguously identify the transferred packet and interpret the payload in the data field. [ASA15b, p.84], [PZ16, p.21]
Two packet types exist. The first one is the 'Control Transfer Objects' (CTO). This is used for protocol commands (CMD), responses (RES), errors (ERR), events (EV) and service requests (SERV). The second packet type is called 'Data Transfer Object' (DTO). It is used for transferring synchronous data: 'Data Acquisition' (DAQ) and 'Data Stimulation' (STIM). The figure 3.2 on page 54 shows an overview of the communication flow between master and slave. The arrows in the figure between master and slave show the communication direction of the different XCP packet types. The exchange of CTO commands is described in more detail in section 3.5 'Commands'. The synchronous data exchange with DTOs is described in section 3.6 'Data exchange'.

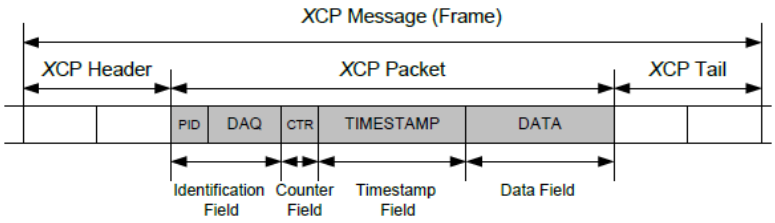


Figure 3.3.: XCP packet [ASA15b, p.84]

Identification field The content of this field depends on the packet type. It is the first field of a XCP packet
In case of a CTO packet, a PID number is used as identification of the XCP packet. The

PID is used by the master and the slave to determine, which packet was send and how to interpret the data field payload. PID numbers are reserved for every single XCP command. The commands from master to slave use the PIDs from 0xC0 to 0xFF. The slave uses the PIDs 0xFC to 0xFF to respond or inform the master.
[PZ16, p.21], [ASA15b, p.85]
For DTO packets the identification field is used to determine to which DAQ list and ODT in this DAQ list the value in the data field belongs to. Packet identification for synchronous data transfer is further described, after introducing the principles of DAQ and STIM data transfers in section 3.6 'Data exchange'.[PZ16, p.21][ASA15b, p.85]

Counter field This field can optionally be used by the a DTO packet for DAQ and STIM synchronous data transfers. A counter is used for DTO CTR mode. It is located after the identification field[ASA15b, p.88]

Timestamp field DTO packets can optionally contain a timestamp field. This field is directly located after the identification field or after the counter field, if present. The slave must support timestamps and run a clock to use this feature. A clock is a free running counter (FRC). The timestamp provides an additional time information for a synchronously send value. The master can thereby correlate the value of a parameter with other values and values from other parameters. The time it takes to transfer a value from slave to master is of no relevance anymore, as the exact time of the measurement is thereby known.[PZ16, p. 21],[ASA15b, p.89]

Data field The last field is the optional data field. The payload depends on the packet type and the specific command. CTO packets contain parameters defined by each command individually. DTO packets contain the value of the synchronous measurement. In case of DAQ this is a value from the slave, which is send from the slave to the master. For STIM it is a value from the master to the slave. An example for a specific payload of a data field is given in section 3.5 'Commands' and section 3.6 'Data exchange'.[PZ16, p.22], [ASA15b, p.91 f.]

3.4. Transport layer

Following transport layer protocols can be used for XCP version 1.3:

- CAN / CAN FD
- Ethernet (TCP/IP, UDP/IP)
- FlexRay
- SxI (SPI and SCI)
- USB

[ASA15b, p.9]
This section will describe the Ethernet transport layer for XCP, because Ethernet is the most future proof in the automotive sector compared to the other protocols. For high performance measurements, XCP on CAN / Flexray, are no longer an adequate technology [A15, p.2]. Additionally, Ethernet can be used for virtual ECUs, like a PC based ECU simulation [PZ16, p.57].
XCP on USB has no relevant market share [PZ16, p.60]. Nevertheless, XCP on USB could be used for ARM / Arduino development boards for university projects. OpenXCP currently supports Ethernet. For the future it is planned to support USB for the mentioned reason.
Every transport layer protocol embeds the XCP packet and adds a specific XCP head and a specific XCP tail. Header, packet and tail are called XCP frame. In the case of Ethernet the tail is empty. The header consists of a control field, starting with a 'Length' (LEN) entry, followed by a 'Counter' (CTR) entry. LEN contains the number of bytes of the XCP packet. CTR is a counter for XCP packets send by the master and the slave. Master and slave have a independent counter, which they increment on every XCP packet they send. The CTR entry is useful to recognize packet loss on the application layer. This is useful if UDP/IP is used, which does not handle packet loss. In contrast, TCP/IP is a reliable protocol with built in error checking. Though, UDP is mostly sufficient for measuring and calibrating. If it comes to measuring a signal sporadic packet loss is not a problem, because a new measurement value is acquired and transfered shortly after. It only produces a very small measurement gap. Detecting the packet loss on application level and resending the packet is not necessary, because the value is already

outdated anyway. When it comes to calibration of parameters, the calibration action can be verified by requesting and comparing the value of the parameter on application level. On the other hand, UDP is best suited for high performance measurement, because of its connection-less and low overhead qualities. TCP should be used, if every measurement value is of great importance. [ASA15a], [PZ16, p.49 ff.]
A UDP packet can contain one or more XCP frames, as long as a XCP packet does not cross a UDP packet boundary. Transferring more XCP packets in one UDP packet reduces the overhead and increases the network throughput. The maximum UDP packet size must be respected. This depends on limits of the underlying IP and the Ethernet protocol.To connect master and slave an IP address and a port must be configured for each. Each XCP transport protocol layer standard defines special XCP commands (CMDs) that are unique for the transport protocol. [ASA15a], [PZ16, p.57 f.]

3.5. Commands

This section describes the different CTOs and how master and slave exchange them.
XCP is a single master, multi-slave protocol. It should be noted, that the master cannot broadcast messages to slaves. The master must establish a dedicated connection to each slave. [MPC08, p.4]
Concerning CTO messages, the master always initiates the communication with a command (CMD) and the slave answers with a positive response- (RESP) or a negative error-message (ERR). A command (CMD) is identified by an unique PID and contains command specific parameters. The maximum number of bytes, that can be contained in a CTO message is defined by the MAX_CTO parameter. A positive response from the slave always contains the PID 0xFF. Optional response parameters can be added, depending on the command which is answered to. [PZ16, p.22]
A negative response contains the PID 0xFE, an error code and optional parameters. Twenty different error codes are defined by the XCP standard [ASA15b, p.228]. Besides that, the standard also defines the error handling procedure for every single command. A request of a master must be replied by a response of the slave. For every command send by the master a timer is set. If the slave fails to respond, the timeout of the master reaches its maximum and a timeout error is triggered. After that, for example, the packet maybe retransmitted or the session is terminated. For every command the

XCP standard defines an error handling procedure. Refer to XCP base standard for details [ASA15b, p.227].
The slave is also capable of sending two special CTO message types. The first is called EV. This is used by the slave to inform the master about an asynchronous event. Such an event can be the failure of a functionality of the slave. This event should not be confused with events from synchronous data transfers (DAQ). The PID of an EV packet is 0xFD, followed by event code definied by the XCP standard an an optional event information. The second special command is called SERV and is used by the slave to request execution of a certain service by the master. The PID is 0xFC followed by defined service request code and optional service request data. [PZ16, p.29]
Three different communication types are supported to transfer CTO messages. In the standard mode the master sends a single request and the slave answers with a corresponding single response. A more efficient mode for commands, which transfer large amount of data (e.g. upload, download) is the optional block transfer mode. The master can send several commands, before the slave answers with one response. Also the slave can send several commands as a response to one request of the master. The third mode is called interleaved mode and also helps to improve performance, but is of no practical relevance. [PZ16, p.24]
XCP is scalable to project requirements and hardware limitations. Mandatory commands must be supported by every slave.
Commands are grouped in categories:

- standard (connect, disconnect, information exchange, measurement)
- calibration
- page switching
- basic data acquisition and stimulation (DAQ, STIM)
- static data acquisition and stimulation (DAQ, STIM)
- dynamic data acquisition and stimulation (DAQ, STIM)
- non-volatile memory programming (flash programming)
- time synchronization (advanced time synchronization)

[ASA15b, p.97 ff.]

Each command category has mandatory commands. If a category is wished to be supported, than the mandatory commands of this category must be supported at least. Other commands of the category are optional. A list of all commands with their individual PID can be found in [ASA15b, p.97]. [ASA15b, p.264] shows examples on how commands must be exchanged between master and slave. The communication flow must follow a defined pattern of commands by the master and response of the slave. The supported commands by a slave are defined in the slaves A2L file. The master can also determine, if a command is supported by the slave: If a unsupported command is send to the slave, it replies with a 'ERR_CMD_UNKNOWN' response. The master can than update the A2L file. [PZ16, p.25 f.], [ASA15b, p.97 ff.]

3.6. Data exchange

3.6.1. Measurement and calibration with CTO

Measurement and calibration can be preformed based on CTOs. This method is referred as polling mode. To measure a parameter the master sends a request to the slave. The command used for reading the value of a memory address from the slave is called 'SHORT_UPLOAD'. The slave executes and answers this command by sending a positive response including the value of the signal. Each measurement must be polled separately and sequentially. This sums up to two messages per measurement: request and response. This is a disadvantage, because it increases the data traffic on the transport layer dramatically. Another disadvantage is, that multiple measurements can not be correlated to each other. Moreover, the measurement includes no timestamp and does not correlate with internal ECU events. Also the packet transfer time from slave to master can distort the measurement result.Therefore, the polling mode can not be used to acquire data at a specific ECU event. These disadvantages can be solved by the DAQ synchronous measurement mode. [PZ16, p.33] [ASA15b, p.28 f.]
The standard way to calibrate an ECU parameter is done by CTO commands. The master instructs the slave to set a given value at a specific memory address. Therefore the master sends three commands in a defined order to the slave. The first command 'SET_MTA' contains the memory address of the parameter. Next, the 'DOWNLOAD' command sends the new value to the slave, including the number of bytes of the new

value. To verify, if the calibration succeeded, the master requests the new value of the parameter from the slaves memory. Actually, this is the same process used for measuring signals. Again, the command 'SHORT_UPLOAD' is used for measuring the value of the parameter. The received value is than compared with the desired value. Every command from the master must of course be followed by a response from the slave. This adds up to six messages for one calibration. [ASA15b, p.268]
The main disadvantage of this calibration method is, that a change of a parameters value is not synchronized to internal ECU events. This is solved by the STIM synchronous calibration mode [PZ16, p.42].

3.6.2. Synchronous measurement and calibration with DTO

Data Transfer Objects (DTO) are used to exchange measurement and calibration data, which are synchronous to internal events of the ECU [PZ16, p.32 ff.].
The DAQ mode for measurement is divided into two phases. In the initial phase the master configures the slave. The master informs the slave about signals that are going to be measured and the affiliation of the signals and ECU events. The second phase is started by the master sending a command to begin the actual measurement. The slave then sends the values of the measured signals to the master, without further requests by the master. The data collection and the transfer are triggered by the occurrence of the ECU events. The slave continues to send data until the master stops the DAQ measurement by command. [PZ16, p.32 ff.], [ASA15b, p.29]
The acquisition of measurement values is preformed by the XCP driver, when the slave completes the execution of an algorithm and triggers the corresponding event. This ensures that all computation cycles are finished before the values of the signals are collected. The slave retrieves the values of the measurement signals from the ECUs memory and stores them to a buffer. Each event has one or more signals attached to it. One signal is only connected to one event and not to multiple events. After the measurement values are acquired, the slave sends them as a DAQ DTO message to the master. Therefore all values correlate to the event. [PZ16, p.32 ff.] [ASA15b, p.29]
An event can be cyclic or acyclic. Cyclic events can be based on timers. For example an event is triggered every 100 ms by a timer-interrupt. A cyclic event can also be angle-synchronous, depending on the rpm of an engine [PZ16, p.35]. An acyclic event is

for example an user input. The configuration step of a synchronous DAQ data transfer

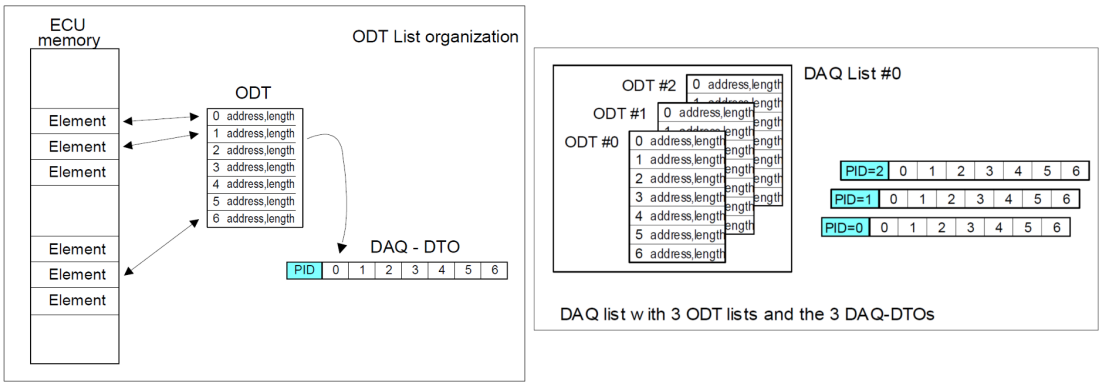


Figure 3.4.: ODT- and DAQ- list organization [ASA15b, p.13, p.15]

is done by the master. The user selects which signals are connected to an event. This information is saved inside a DAQ list. Each event has its unique DAQ list. A DAQ list contains entries for the measurement signals memory address and size in memory. These entries are stored in Object description tables (ODTs). A DAQ list can have one or more ODTs. [ASA15b, p.13 ff.]

The slave sends one or more measurement values connected to an event in a combined message. This reduces data traffic. During configuration step, the master informs the slave how to compose the bytes in the message. The before mentioned Object description tables (ODTs) define the structure of a message. An ODT includes a reference to the slaves memory (RAM) for each measurement signal of an event. An ODT entry contains the starting memory address and the size of the measurement signal. This called an object or ODT entry. The index of an ODT entry refers to the position in the DAQ DTO message. The relationship between ECU memory, ODT and DAQ DTO message is shown in figure 3.4 (left). [ASA15b, p.13 ff.]

During measurement the slave retrieves the individual objects data from the ODT and combines them into packets. The packets are send to the master. The master knows how to interpret the message, because the master defined the structure during configuration. The master extracts the objects values and assigns them to the corresponding measurement signals. However, the transport protocol has a maximum number of bytes it is able to transfer. The maximum size of a DTO message is defined by MAX_DTO. Therefore, a DAQ list must contain more than one ODT, if the number of measurement

objects exceeds MAX_DTO or MAX_DTO_ENTRIES. The relationship between DAQ lists and ODTs is shown in 3.4 (right). [ASA15b, p.13 ff.]

DAQ / ODT summary: [ASA15b, p.13 ff.]

- One DAQ list per event. DAQ list is triggered by event.
- DAQ list contains ODTs.
- ODT entries map the slaves memory to the data transfer object (DTO)

DAQ list can be configured in three different ways: static, predefined and dynamic. Predefined configuration is not discussed, because it is has no practical relevance.

DAQ static [PZ16, p.40]

- Maximum number of ODT entries per ODT of a DAQ list is specified
- The number of ODTs per DAQ list is specified
- Only the framework is defined, not the measurement signals itself (ODT entries)
- Example: DAQ list 0 is defined to a 10ms event and can contain max. two ODTs
- Specification is determined with the download of the program to the ECU

DAQ dynamic [PZ16, p.41]

- Maximum number of ODT entries per ODT is dynamically configurable
- The number of ODTs per DAQ list is dynamically configurable
- Only the total memory space for the DAQ lists is specified (max. DAQ lists)
- DAQ configuration is determined by the master during runtime with CTO commands (setup of DAQ measurement)
- Benefit: Master can dynamically increase the number of measurements per event

STIM STIM is used for event synchronized calibration. It uses DTOs instead of CTOs to exchange calibration data. The A2l file contains the information about the events, which the master can synchronize to. The master sends the calibration values in packets. The approach is the same as described before with the DAQ list. The slave must know the packet structure to extract the correct calibration value for a parameter. [PZ16, p.42]

DTO packet addressing Compared to CTO packet addressing, DTO packet addressing is slightly more complicated. For CTOs it is sufficient to use an unique PID to identify a packet. For DTOs two identification types exist: 'absolute ODT number' and 'relative ODT number and absolute DAQ list number'. Every DAQ list contains a number of ODTs. The first ODT in any DAQ list starts at index 0, the second ODT at index 1 and so on. Therefore, an ODT number is relative and not absolute over DAQ list boundaries. [ASA15b, p.85 ff.]
The DTO packet addressing method 'absolute ODT number' maps the relative ODT index number to an absolute ODT index number. The goal is to make ODT numbers unique across all DAQ lists during the measurement. The XCP base standard provides this formula [ASA15b, p.86]:

$$\begin{aligned} &absolute\_ODT\_NR(ODT\,i\,in\,DAQlist j) = \\ &FIRST\_PID(DAQlist\,j) + relative\_ODT\_NR(ODT\,i) \end{aligned} \tag{3.1}$$

A DTO measurement packet starts with the FIRST_PID. This PID has the number j from the DAQ list. The second packets PID has the number j+1. The third packet has the PID j+2. The number that is added to the FIRST_PID depends on the relative ODT number of the particular DAQ list. FIRST_PID plus relative ODT number must not overlap with the FIRST_PID of the following DAQ list. [PZ16, p.43]
The DTO addressing method 'relative ODT number and absolute DAQ list number' is an alternative to identify DTO packets. The identification field contains a PID and an absolute DAQ list number. The PID is a relative ODT number from the DAQ list. [ASA15b, p.86]
ODT messages can contain a timestamp field for correlating different measurements to each other. The XCP standard defines several mechanisms to correlate the timestamp of one measurement data to a timestamp of another measurement data. One method

is, that the slave includes timestamps based on the slaves clock. The timestamp information is included in the first ODT of every DAQ event of a DTO message. When an event is triggered in the slave, the XCP driver acquires the time of the event and collects the value of the corresponding measurement signals from memory (RAM). The master receives the DTO packet including the timestamp, but does not know how the timestamp correlates with other clocks. Therefore, the master can request the slaves clock value by sending the GET_DAQ_CLOCK command. The slave then answers with the current value of its internal clock. Hence, the master can calculate the offset of the slaves clock to the masters own clock. This can be done, because the master knows the point in time for the request and the slaves reply. [PZ16, p.45 f.].
A more precise clock correlation method is called 'Multicast'. The master requests the clock of multiple slaves at the exact same time. Each slave saves the time, when the request command was received. The slave then replies to the master by sending the recorded arrival time of the request command. The master can calculate the correlation between the masters clock and all slave clocks. [PZ16, p.46].
Another method is called 'Grandmaster Clock'. The slaves clock is synchronized with a grandmaster clock by the IEEE 1588 Precision Time Protocol. [PZ16, p.47]

3.7. Optional services

A brief introduction is given to non-mandatory services of XCP. These services are only implemented by the slave, if the project requires this. [PZ16, p.61] lists following services:

- Memory page switching
- Save memory pages
- Flash programming
- Automatic slave detection
- Block transfer mode
- Measurement during slaves power-on
- XCP security features

Memory page switching In the perspective of XCP the slaves memory is a continuous physical space. Physically the memory layout is described by sectors. Logically the memory is described by segments. A segment is divided in a number of pages. Segments hold the information, where calibration values are stored in memory. 'The PAGES of a SEGMENT describe the same data on the same addresses, but with different properties e.g. different values or read/write access' [ASA15b, p.42]. The slave can only access the data of one page at a time for this segment. This is called the 'active page of the ECU'. The XCP master can also only access the data of one page at a time for the same segment. This is called the 'active page of the XCP master'. The master can switch the active pages independently for every segment. The slave can not switch pages by itself. [ASA15b, p.44] The advantages of switching pages is, that complete sets of parameters can be changed at once. This is beneficial for comparing the behavior of an algorithm before-and-after changing the parameter set. Another advantage is that the plant (e.g. engine) is always in a stable state. Normally many parameters must be configured to gain the desired outcome of a plant or system. If this is done in single steps, the plant can get into an unstable state.The calibration would be incomplete. This is prevented by setting the parameter set active as a whole at once. [PZ16, p.62]

Save memory pages The slave can be commanded by the master to save the calibration data into non-volatile memory, e.g. flash. This is called 'calibration data page freezing'. [ASA15b, p.44].

Flash programming XCP can also be used to flash an ECU during development phase. A so called 'flash kernel', a executable code, is sent to the slaves RAM. It handles the XCP communication during the flashing process. According to [PZ16, p.63 f.] the flash process can be divided in three steps:

1. step: prepare the flash process (check if new data can be flashed)
2. step: preform flash process (new data is sent to slave)
3. step: post-processing (integrity check of flashed data, e.g. checksum)

Automatic slave detection The master can request information about the slaves XCP configuration and the limitations of the slaves XCP implementation. Several CTO commands are provided for retrieving this protocol-specific information. Basic information can be obtained, for example communication options and current status of the slave. Also DAQ specific information can be requested. For example, this includes general DAQ information, DAQ configuration and provided ECU events. A more detailed description can be found in [PZ16, p.65].

Block transfer mode As described in last paragraph in section 3.5, block transfer mode improves communication performance. This is useful for transferring large amounts of data between master and slave. Several messages can be sent in a block, without the need to acknowledge every single message individually. [PZ16, p.66]

Measurement during slaves power-on Usually measurement is only possible during the normal runtime of the slave and not at the startup of the ECU. The master must first configure the measurement, before the actual measurement and data exchange can start. This configuration is generally saved in the slaves volatile RAM. For being able to measure signals immediately after slaves power-on, the measurement configuration must therefore be saved in in a non-volatile memory. In conclusion this process can be divided in three steps. First, the master configures the slave for DAQ measurement. Secondly, the slave stores the configuration to EEPROM or flash memory. Then the slave reboots and loads the existing DAQ measurement configuration from the non-volatile memory and immediately starts transferring measurement values to the master.
[PZ16, p.67]

XCP security features XCP provides a 'seed and key' mechanism to prevent unauthorized access to the ECU by XCP. First the slaves sends a random number (seed) to the master. The master calculates a key number from the seed and sends it back to the slave. The slave individually calculates the expected number. Then the slave compares the masters key with the slaves calculated result. If the numbers match, the slave and the master have used the same algorithm for calculation and the slave allows the XCP connection. The downside of this method is, that if an attacker knows the algorithm

in use, he can of course calculate a matching key from the seed. Therefore, XCP functionality should be deactivated after development and calibration phase. Nevertheless, with a special combination of XCP commands the slaves XCP driver can be activated again. This is sometimes even the case in production vehicles. [PZ16, p.68]

3.8. Data model for ECU

The ASAM MCD-2 MC standard (ASAP2/A2L) describes a data model for ECU measurement and calibration. The ASAP2 data model is a non-vendor specific standard for online measurement and calibration XCP-tool-chains. ASAP2 has the file ending A2L and is based on a non-XML ASCII text-format.
An A2L file describes all information of a slave needed by the XCP master tool to measure and calibrate the ECU. The A2L file is the single source for a master tool. Therefore, it includes description of measurement signals and calibration parameters, conversion rules and characteristic maps, events, communication device interface (transport protocol). These A2L description groups will now be described briefly. [ASA15c]
The calibration engineer uses symbolic names, provided by the master tool, to calibrate an ECU. On the other hand XCP uses addresses to access ECU internal variables. Therefore, A2L includes the information, which maps an address to a symbolic name. [PZ16, p.72 f.]
Also the attributes of the variables are stored in an A2L file. For example the minimum/maximum allowed value of a variable or a conversion rule, that maps raw values to physical values.
Characteristic maps help to resolve the relationship between a ECU input value $'x'$ and a desired result $'y'$. For example the relationship between $'x'$ and $'y'$ is expressed by the function $'y = f(x)'$. The result $'y'$ can be looked up in a characteristic map for the input $'x'$, without the need of complex computation during runtime. [PZ16, p.73]
Synchronous DAQ/STIM information and events of the slave are also described in the A2L file.
Furthermore, an A2L file stores information for communication parameters, for example the slaves transport protocol and its parameters. In case of Ethernet: IP address and port. [PZ16, p.73 f.], [ASA15c]
It can happen, that a description in the A2L file does not match the actual configuration

of a slave. In this case the master has the ability to update the A2L information. This is done with XCP CTO commands. Commands for retrieving the slaves configuration settings are described in section 3.7 'Automatic slave detection'. [PZ16, p.76] Excerpts of an example A2L file, generated by OpenXCP, can be seen in listing 5.1 on page 66. A more in depth description of ASAP2/A2L can be found in ASAM MCD-2 MC standard [ASA15c].

4. Architecture

4.1. System architecture

The following chapter will explain the system architecture. The system architecture can be divided by software or by hardware boundaries. The straightforward approach is to distinguish master and slave by hardware boundaries. The master is a software which runs on a normal computer. On the other hand the slave software runs on an embedded hardware platform. In this case on an Infineon Aurix TriCore ECU. The slave software could also be simulated on a computer hardware instead of running on a dedicated ECU. In this context the term slave describes the software and the ECU hardware in one term. The simulation of the ECU software on a computer is often needed during early stages of development. However, the use case for a measurement and calibration tool chain is to support product development on the target hardware. For this reason, as one would expect, the slave software is implemented on an ECU hardware.
The goal of OpenXCP is to measure internal processes in the ECU firmware and to calibrate the system during runtime. Therefore, a XCP driver component was implemented on top of an existing firmware. For the show case of this project an example project for the Infineon Aurix TriCore was used. For the communication between master and slave the XCP protocol must be implemented in both entities. A XCP driver gets implemented on top of the existing ECU firmware. As XCP is an application layer protocol, referring to the OSI model, one also has to define the transport layer. The underlying transport layer is specific to the project requirements. In this project the transport layer for XCP is Ethernet with UDP/IP. Ethernet is chosen, because it is increasingly popular in the automotive sector. Also Ethernet ports are common on development boards.The given automotive development ECU board from Infineon is no exception.

The master is the software that controls the slave. The slave reacts to the XCP commands send by the master. This is a one-to-one connection. The OpenXCP master can only control one slave and a slave can only be managed by one master. To keep the XCP driver simple for low resource ECUs, the master manages the XCP measurement and calibration session. In order to setup a working session the master has to fulfill several tasks. The tasks and its corresponding software components within the master are described below.
The master OpenXCP software can logically be subdivided into XCP-interface, transport layer interface for the slave communication, ELF/Dwarf and C-source parser, export and saving of project settings, record of slave data and an user interface. Looking at the architecture from a users view of perspective: Starting a measurement and calibration session, a user configures the transport layer interface between master and slave. Next, the XCP-interface is configured to meet the given project requirements. Being able to access the slaves ECU memory during runtime by the master, memory layout information must be available beforehand. This is done by parsing the ELF-file. The information from the ELF file is flashed into the memory of the slave. By parsing the ELF file and the included DWARF debug informations the master can extract the memory address, size, name and so on of all variables which are accessible by the master.
To enrich the measurement and calibration process a set of further information can be added to each variable that will be monitored. This set of information is defined by the XCP standard. The user can either input this data via the GUI or by using special comments in the source code of the slave firmware. The later approach is chosen by OpenXCP. The C-source code is parsed for defined XCP comments containing further information for a variable. This is for example, human readable description, calibration or measurement setting, minimum and maximum legal value and so on. Combining all the input from ELF/Dwarf and source code parsing gives the starting point for XCP session. The user can than choose which variables are part of the current workspace. The configuration of XCP, transport layer and variables in workspace can be saved to a JSON file for further use. This is useful, if the OpenXCP program is closed and work is continued at some point in the future. The project save-file in JSON format will be read by the next time OpenXCP is started. Thus, the user can continue work on the previous basis. The project data can also be exported to an ASAP2 standard file in a A2l file format. This is necessary to be compatible with the standard and other XCP master

software, like CANape [Gmb18]. Due to the fact that ASAP2 is an automotive standard, there are none open source production ready libraries for writing and parsing A2L files. Therefore, in the scope of this project, JSON was used as internal project file. A2L files are generated by using a template file and enriching it with the workspace settings. For saving values and timestamps of each monitored variable the data is saved in a data structure inside the master. This data can be exported to a comma-separated-values-file (CSV) at the end of a XCP-recording session. This CSV-file can now easily be further processed. The data can either be plotted or evaluated in another way. Useful programs for this task are for example MATLAB [The18]. The system architecture is shown in the figure below.

The core of the slaves XCP-interface is the free-to-use XCP basic driver from Vector GmbH. This driver includes a state machine to manage the measurement and calibration process, it supports mandatory XCP commands and functions. It also provides an interface for defining project specific functions. These functions must be implemented depending on the requirements of the project, hardware and supported transport layer. In this case the Infineon Aurix TriCore ECU with Ethernet and UDP/IP. The UDP driver was added to the XCP driver by implementing the required callback functions. The UDP driver uses lightweight IP (LwIP) as a network stack. The rest of the XCP driver is configured by preprocessor directives. As a basis for time driven XCP-events a timer interrupt is used to trigger XCP-events. The show case software on the ECU is using the Infineon framework. All software on the ECU is implemented in C programming language.

The OpenXCP master is implemented in C++11 using the Qt 5.9 framework. The Qt modules core, gui and network are used for the corresponding tasks. Besides Qt, OpenXCP uses the library libelfin for parsing the ELF file and its Dwarf data structure. For writing the CSV record file the qtcsv library is used.
Figure 4.1 shows the system architecture. The figure shows the XCP master software, the XCP slave Aurix TriCore and a third party software. In dotted lines a second XCP slave is shown. This illustrates how highly versatile XCP can be used for a variety of ECUs and transport layers.

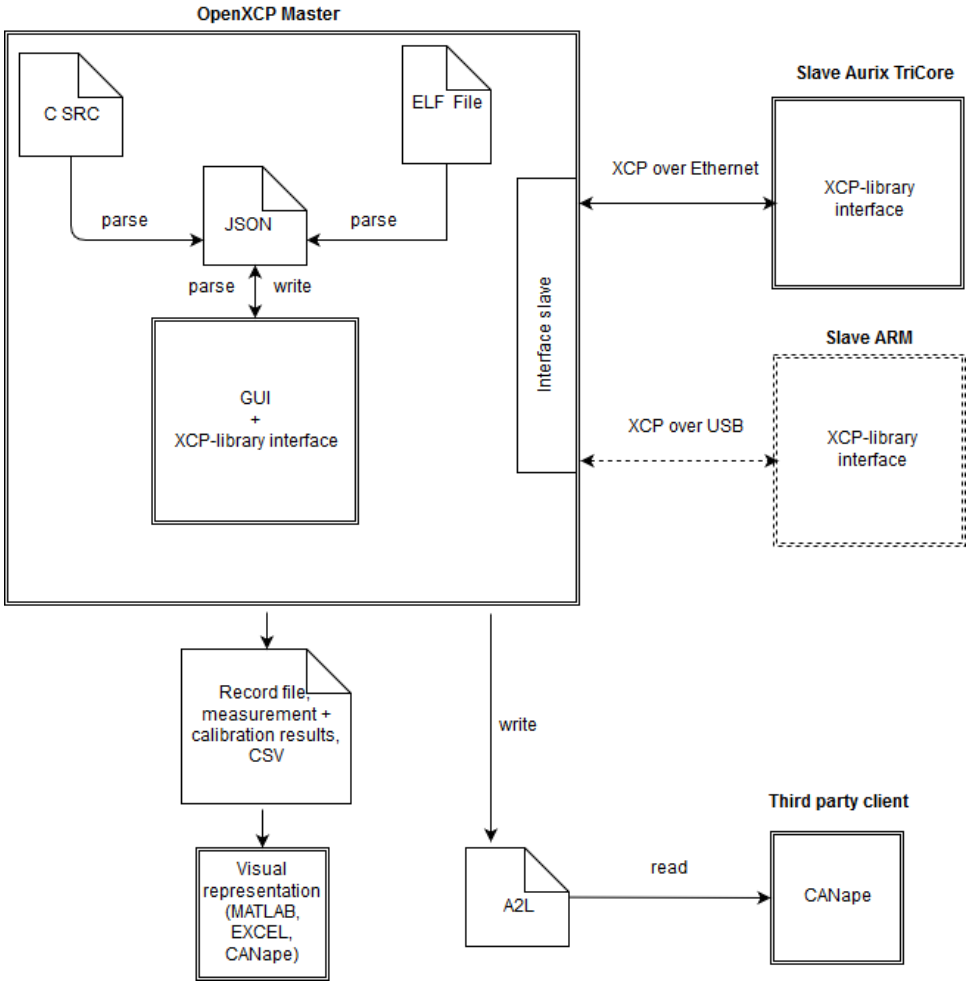


Figure 4.1.: System architecture

4.2. Master

4.2.1. Top level overview

This chapter will describe the software architecture of the XCP-master. The software is subdivided in different modules. Using a object oriented programming language these modules are mapped to classes. One logical module consists of a number of tightly bounded classes. Figure 4.2 shows an overview of the top level classes. The benefit of using an object oriented approach for the software architecture is that project requirements can easily be transfered to source code. The tasks of the software are encapsulated in classes. This means that the implementation of the task is hidden from the view of outside objects. Well defined interfaces between tasks and their corresponding classes allows to create a modular software design. Therefore, the source code gets manageable and objects can easily be reused. Overall the maintenance of the code gets easier. Also inheritance helps to abstract a problem in smaller easier to understand code. Inheritance together with polymorphism reduces the lines of copied code and therefore the software has a reduced footprint. Which again results in more maintainable software. The modules of OpenXCP are an abstraction of the project requirements regarding a XCP-master software. A XCP-master must support the XCP-protocol and manage a slave for measuring and calibrating its internal memory. To achieve this goal several steps have to be undertaken. The top level classes are introduced representing the core of the software architecture.
The *Backend* class is the core of the software architecture. It controls the program flow. The *Backend* is the so called business logic in the model-view-controller design pattern. The model part of the MVC pattern is divided in two main classes. One is responsible for storing the data and the other for controlling the program flow and the corresponding algorithms. The data-storage class is called *Model* and the later *Backend*. The user interface in OpenXCP is a graphical one. It is represented by the *MainWindow* class. The GUI is managed by the *Controller* class. The *Controller* also has direct access to the data in the *Model* to represent the data in the view. The GUI shows the user all the actions that can be triggered. If an action is chosen the controller gets informed about it. Depending on the chosen task, the controller changes the view directly or informs the back end. Tasks that only affect the representation of data in the view are handled by the controller in place. Data gathering, parsing and algorithmic jobs are

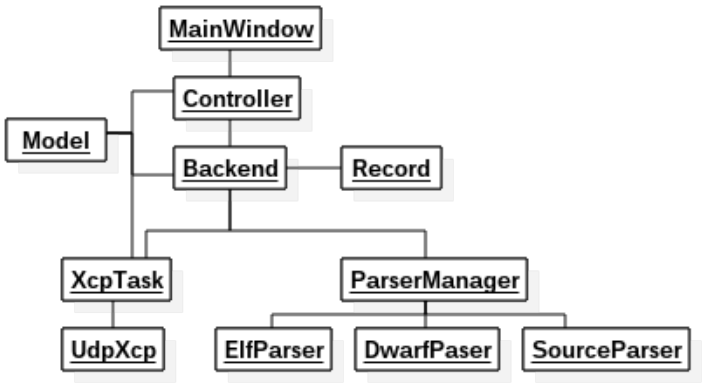


Figure 4.2.: Overview

delegated to the back end. The back end then runs the appropriate action and saves the result in the model data structure. The controller gets informed by the back end about the state change and the result of the computation. Now the controller reads the latest data from the model and updates the view.
The main task of the back end is delegating and controlling the specific modules in the software. It is also the interface between all business logic classes and the controller. Especially it is an interface between the controller and the *XcpTask* class as well as an interface between controller and *ParserManager* class. The later is responsible for managing the three parsers in OpenXCP. The first parser is called *ElfParser* and parses the image file from the slave (ELF). Next, the *DwarfParser* also parses the slaves ELF image file. In this case the parser extracts debug information about the slaves memory by reading the DIE tree. Finally, the *SourceParser* class is responsible for reading special XCP-comments embedded in the C-source code of the slaves code base.
The before mentioned *XcpTask* class is necessary for managing the XCP session between master and slave. The *XcpTask* class internally uses the finite state machine design pattern. In the appendix B a UML diagram illustrates the state machine. In each state different XCP commands are send to the slave or received from it. *XcpTask* coordinates the sequence and timing of the commands. Also the payload of each command is composed by it. This all must be done by meeting the rules and requirements of the XCP standard. The *XCPTask* state machine has following states: CONNECTED, if a connection is established with the slave. RUN, requesting, receiving, modifying and

recording of the slaves monitored memory. STOP, for ending a measurement and calibration session and saving the data to a file. DISCONNECTED, if no connection is established to the slave or the master closed the connection.
The XCP standard supports different kinds of underlying transport and network layers for encapsulating the XCP payload. As described before, in this show case setup, Ethernet with UDP/IP was used. Nevertheless, the architecture of OpenXCP is designed to support any other technology. Such as Ethernet with TCP/IP, USB or CAN and so on. This is achieved by separating the XCP protocol layer and the transport layer implementation. This ends in a well defined interface and reusable code.For now, OpenXCP has implemented the *UdpXcp* class to connect to the slave via UDP in an IP-network. Especially the extension of a TCP/IP and USB transport layer class implementation is well prepared in the OpenXCP source code and architecture. The *UdpXcp* class provides public send and receive methods. The send method is called by the *XcpTask* to encapsulate a XCP packet in a UDP packet and sending it to the slave. On the other hand, the receive method handles incoming UDP packets and informs the *XCPTask* about it, providing the XCP payload.
Another top level class is *Record*. This class is used for saving recorded data from the slaves monitored measurement and calibration variables. The class is just a data structure for convenient serialization of the records to a CSV file.

4.2.2. Model

As mentioned in 4.2.1 before, the *Model* class holds all the data of the software. The data structure of the model is separated in a number of classes. This is beneficial, because the software architecture can be divided in smaller portions to reduce the overall complexity. Figure 4.3 shows the classes related to the model. The objects of the classes are mainly shared in the software by pointers. This avoids unnecessary copying of memory intensive objects. In fact the *Model* class purpose is to hold all the references to the task specif data classes. This has the benefit, that business logic classes, like the *Backend*, only have to include this single class to get access to the softwares data. Hereinafter the data classes are briefly described. The *EthernetConfig* class is used to save the users transport and network layer settings. In this case the Ethernet related setup, such as UDP/TCP, ip address and port of the master and the

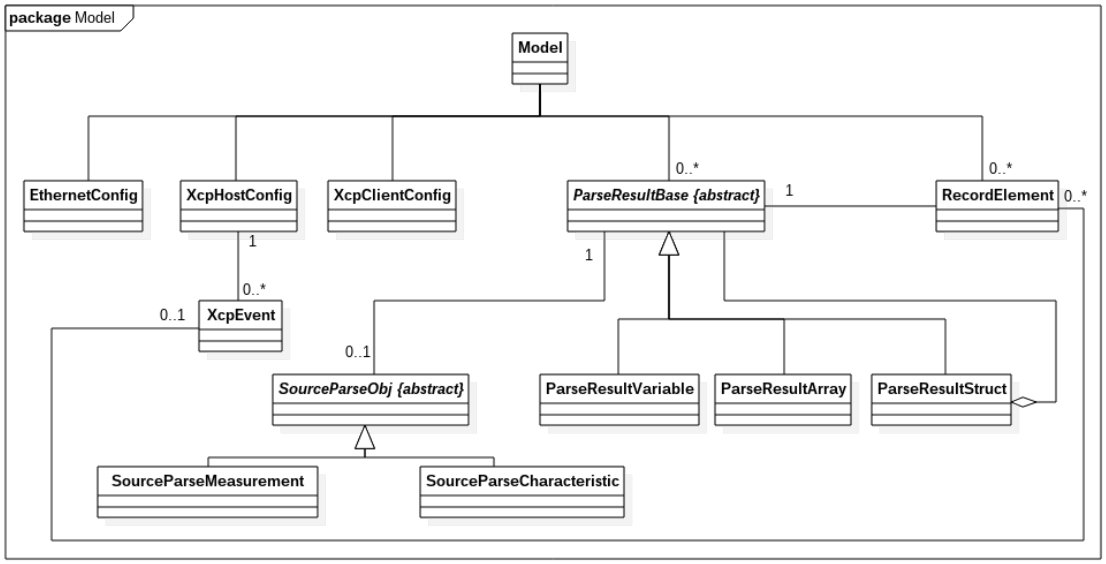


Figure 4.3.: OpenXCP-Model

slave. For other mediums, like USB, a class in the same manner has to be added to the software architecture. Next, the *XcpHostConfig* class is responsible for holding the XCP settings. These settings must match the configuration of the slave. This for example is the byte-order or the maximum size of a XCP packet. Connected to this class is the *XcpEvent* class. Depending on the project a XCP slave can support none or many events for DAQ mode. The *XcpClientConfig* class saves the received settings from the slave. The client settings can be compared with the host settings to see if they match. If they do not match the master can update the XCP host settings or start an error handling procedure. Also the *XcpClientConfig* stores additional information about the slaves internal state and XCP configuration.
The next class is about storing data from the parsing process. The abstract class *ParseResultBase* is the base class for the derived classes *ParseResultVariable*, *ParseResultArray* and *ParseResultStruct*.These classes store the result of the ELF-, DWARF- and source code comment parser. The ELF- and DWARF parser resolves the name of a variable, memory address, size, data type and so on. Subsequently, the parsers resolve and distinguish if a variable is a basic variable, an array or a struct data type. Structs can contain arrays, basic data type variables or again struct members. This is also true

for arrays, but in this stage of OpenXCP only arrays with basic data type variable members can be resolved by the parser. Hence, all three classes share the same basic properties. This is why they have a common base class. For handling the tree structure of the struct variables, the composite design pattern was used. This represents the tree hierarchy of the members of a struct. Another attribute of the *ParseResultBase* class is the *SourceParseObj* class. This abstract class holds the data which was obtained parsing the source code comments by the source parser.
The XCP standard differs between measurement and characteristic. The first is used to monitor the value of a variable and the second for modifying the value of variable. Therefore, both have same base settings, such as a describing comment or the physical unit for the variable. This information is summarized in the base class *SourceParseObj*. Individual options are stored in the respective derived class. This additional information can be obtained by the source comment parser or by manual input by the user via the GUI. This XCP specific data is only present if the user chose to add this extra information. That means that not every variable has this information. Only the ones that are chosen for measuring and calibration. This fact is represented by the software architecture in the way that a *ParseResultBase* instance can have zero or one *SourceParseObj* objects. Resulting in zero or one *SourceParseMeasurement* or *SourceParseCharacteristic* object. A variable can not be monitored and modified in one and the same XCP setup. So either the used pointer points to a *SourceParseMeasurement* or a *SourceParseCharacteristic* object.
The *ParsResultBase* class is used in software for saving parsing results. After parsing the user chooses which variables are going to be monitored or modified. Therefore, the chosen *ParseResultBase* objects get enriched and combined with XCP specific data. This is for example, the polling rate of the master for this variable or the associated event (*XcpEvent* object). This means that every *RecordElement* has one *ParseResultBase*. For the measurement and calibration step each *ParseResultBase* object is resolved in a basic data type variable (*ParseResultVariable*). More precisely, an array containing ten basic data type variables gets split into ten single variables. A struct also gets slit up in basic data type variables. This is because the software only operates on single variables and not on complete data structures. Also source code XCP comments are on single variable and struct member basis. Nevertheless, the logic of the program still remembers if a *RecordElement* belongs to a struct, an array or a basic variable. The model saves a list of all *ParseResultBase* and *RecordElement* objects. Even tough, every *RecordElement*

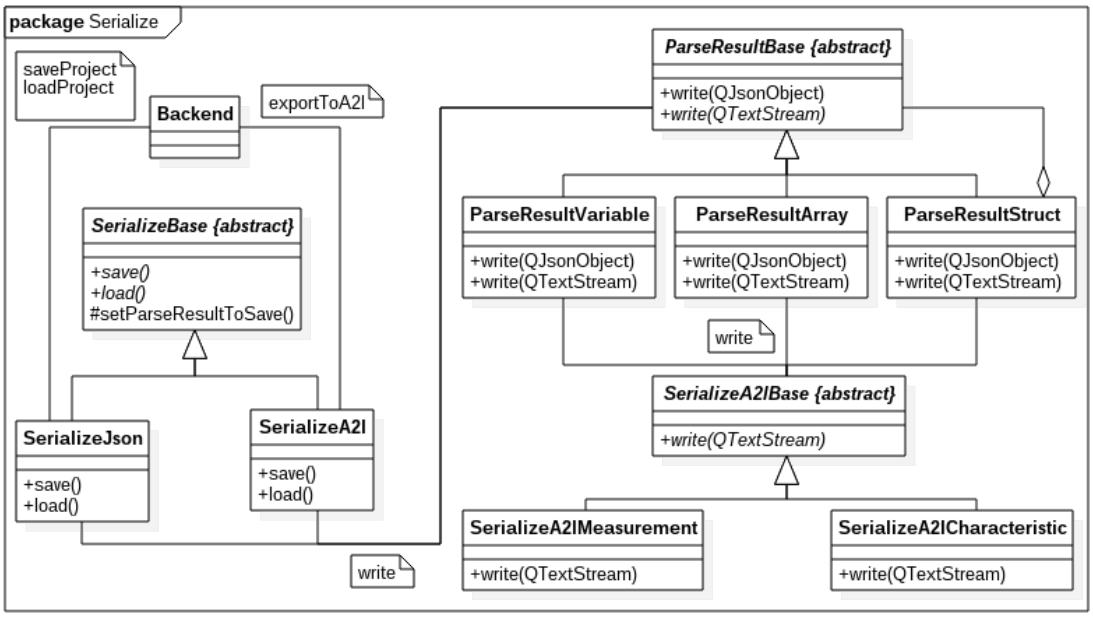


Figure 4.4.: Serialize

object has a *ParseResultBase* member object the data is not just copied and duplicated. A pointer to the original *ParseResultBase* object is stored instead.

4.2.3. Serialization

Also worth mentioning is the software architecture used for serializing project workspace data to a file in JSON or ASAP2 (A2L) format. The UML class diagram in the figure 4.4 illustrates this architecture. The user can choose to save or load the project workspace data to a JSON file. The controller then gets informed by the view about this event and itself passes the event to the backend. The *Backend* class creates a *SerializeJson* object and calls the public save or load method on it. The same is done if the user chooses to export the project workspace data to an A2L file: The *Backend* class creates *SerializeA2l* object and calls the public save method on it. The load method of the *SerializeA2l* class, for reading A2l files, is not implemented. For the time being it is a placeholder.
However, both classes are derived from the abstract *SerializeBase* class. The load method

from the *SerializeJson* class is straightforward. The JSON file is parsed by the Qt build in JSON parser and the data is directly written to the model data structure.
The save operation is of more interest. The save method from both *SerializeJson* and *SerializeA2l* get the project settings, the XCP configuration and the result of the parsing process from the model data structure. This can be achieved by directly accessing the getter methods of the model and its associated data holding classes. However, for the result of the parsing, the *ParseResultBase* class, the data is retrieved by using polymorphism and inheritance. This is because the model holds a list of *ParseResultBase* objects which are stored in a tree data structure. For every ParseResultBase entry its write method is called. The write method gets a *QJsonObject* or a *QTextStream* object as input parameter. The methods add entries to the input parameter, which later gets serialized by the appropriate library from Qt. In case of JSON the *ParseResultBase* write method adds general information to the *QJsonObject* which all three derived classes have in common. The shared data is the additional XCP settings which was retrieved by the source code XCP comment parser. In addition the *ParseResultVariable* and *ParseResultArray* write method add the specific data for variables and arrays. In case of JSON an array does not get split in to single variables. Rather it gets stored as an array containing information such as size and number of elements. In contrast the write method of *ParseResultStruct* calls the write method of its elements. As mentioned before a struct consists of variables and arrays. This means that by polymorphism the appropriate write methods from *ParseResultVariable* and *ParseResultArray* are called and the struct tree is made flat.
Analog to the described procedure the serialization to a A2l file is processed. In this case a *QTextStream* object as input parameter gets passed to the write methods. The write method in the base class is virtual, which means that all three derived classes have to implement their own write method. This is because the measurement and characteristic XCP comment information is written to the text stream object by special A2l-serialization classes. Each derived class of *ParseResultBase* first uses its own write method and than calls either the *SerializeA2lMeasurement* or the *SerializeA2lCharacteristic* class. Structs again are resolved by polymorphism.

4.2.4. Communication between objects

OpenXCP uses the Qt frameworks signal and slot mechanism for exchanging event messages between objects. A signal is an event that is triggered by an object. This signal calls a connected slot method. Signal and slots must therefore be connected to each other. A slot is a normal C++ method. [Ltd17], [LP16]
In OpenXCP the signal and slot mechanism is used to implement the observer pattern between view, controller and back end. For example, a user presses a button in the *Mainwindow* class then a signal gets emitted which calls a slot method in the Controller class. The controller executes the slot method. If the back end is involved in the task, the controller again emits a signal and the back ends slot method is called. The connection between the signals and events is manually done in the *Main* class for controller and back end. Between the controller and the view the connection is handled automatically by the Qt framework. Of course it is also possible that the *Backend* class emits a signal and the appropriate slot method is called in the controller. The back end is also used as gatekeeper between signals from the *XCPTask* object and the controller. Overall the signal and slot mechanism is an easy and flexible to use way for exchanging events, data and messages between objects.

4.3. Slave Aurix

4.3.1. Top level overview

This chapter will describe the software architecture of the XCP-slave. The slave software is based on the 'AURIX TC277x TFT Application Kit Firmware SW V1.1' and uses the C programming language. In the following, the XCP-driver and related modules are described. Figure 4.5 shows an overview of the XCP-driver. The XCP-driver is divided in a XCP-protocol layer and a XCP-transport layer. The ECUs application interacts with the driver by using the drivers API.

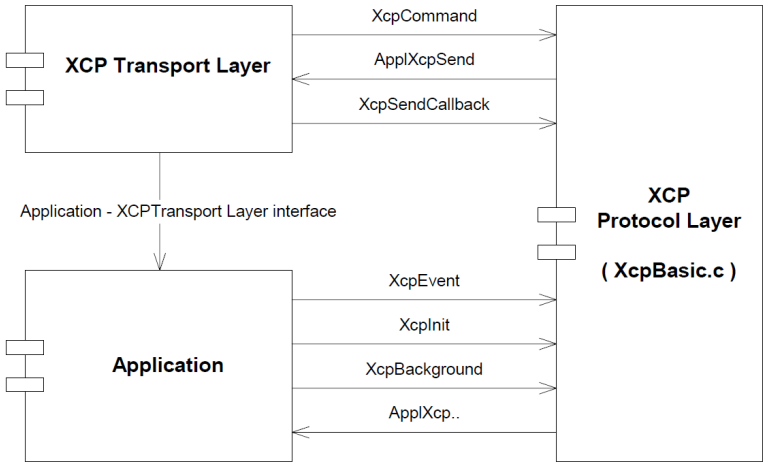


Figure 4.5.: Slave: XCP-driver Architecture [Tri05, p.23]

4.3.2. XCP protocol layer

The protocol layer is provided by Vector Informatik as free-to-use software. It implements the XCP protocol layer. All mandatory XCP commands are supported by this basic version. Vector also offers a professional version of this driver, that supports the complete XCP protocol. For this project the basic version is more than sufficient. Limitations can found in [Tri05, p.68].
The state machine of the protocol driver is fairly simple, because XCP resume-mode is not supported. The slave can either be in state 'disconnected' or 'connected'. After initialization the slave is in 'disconnected' state. In this state the slave remains until it receives a XCP connect-command from the master, an fatal error occurs or the master sends a XCP disconnect-command. If the slave receives a XCP connect-command it changes state to 'connect'. In this state measuring and polling signals and parameters is performed, as well as DAQ configuration. Also the master can retrieve informations about the slave in this state.
The driver is responsible for parsing and interpreting XCP commands. Furthermore the driver is responsible to access RAM memory for measuring and calibrating internal ECU variables. A complete list of all supported XCP functions by this driver can be found in [Tri05, p.11].
The protocol driver provides an API for the transport layer driver and the application.

ECU and application specific functions must be implemented according to hardware requirements. The protocol driver offers stub functions in its API for this. Implementation details are described in section 5.2.

4.3.3. XCP transport layer

XCP supports a variety of transport layer protocols. Therefore, the protocol driver provides three generic interface functions for all kinds of transport layer implementations. The *XCPCommand* function is called by the transport driver, if data was received from the master. The transport driver provides this function the XCP payload, which is than evaluated and interpreted by the protocol driver. With the function call *XcpSendCallback* the transport driver informs the protocol driver about the successful transmission of a XCP packet. In the reverse direction, the protocol driver calls the function *ApplXcpSend* to transmit a XCP packet to the transport driver. This includes the XCP payload. The transport layer is then responsible to embed the XCP packet in a transport protocol specific frame. [Tri05]
In this project Ethernet UDP/IP is used as transport layer. For UDP/IP networking functionality is based on the lwIP stack.

4.3.4. Application

Besides the business logic, the application code must also manage and integrate the XCP transport and protocol driver.
The application is responsible for initializing the lwIP stack for the transport layer and for initializing the protocol layer by calling the *XcpInit* function.
The application also provides a loop, that is called by the main-loop regularly. In this loop time-based XCP events are triggered by calling the *XcpEvent* function with an related event-number. In the same manner, non-cyclic events can be triggered by the application. The protocol layer then handles the XCP event and send a DAQ-DTO message to the slave.
The application provides the protocol layer with CPU time for tasks that have a long calculation time. For example calculating the checksum over a large memory region. This would block the CPU for a long time, if it would not be split up in smaller units.

Therefore, the application periodically calls the *XcpBackground* function to provide CPU time for background tasks. [Tri05]
As mentioned before, the application must implement functions for ECU and application specific XCP tasks. Therefore, the protocol driver provides function stubs in its API. The protocol driver executes the callback functions(*ApplXcp...*), which call the application specific function. A full list of application specific functions can be found in [Tri05, p.40ff.].

5. Development process and software implementation

This chapter will give explanation about relevant implementation decisions and the development process. This also includes examples and analysis of the measuring and calibration process with OpenXCP.
The first half of this chapter focuses on the OpenXCP master. Continued by the slaves implementation. At the end of the chapter a performance test of the complete system is presented. Also the license model for OpenXCP master and slave are defined at the end.

5.1. Master

5.1.1. Choice of programming language

For the following reasons C++11 and the Qt Framework have been selected as basis for OpenXCP master. Refer to the sections in brackets for explanation or requirement.

- C++ is object orientated (see subsection 4.2.1)
- Same language family as slave implementation (C/C++)
- Support of open-source ELF parser (see subsection 5.1.7)
- Multi platform support by C++/Qt (section 2.3)
- Qt is free for open-source projects (see section 2.3 and section 5.3)
- Signal-Slot mechanism of Qt (see subsection 4.2.4)

5.1.2. A2L/JSON

ASAP2/A2L is the standard ASAM data model and file format for XCP. Nevertheless, A2L has several disadvantages for this project. The main disadvantage is, that no production ready open source A2L parser library exists. Implementing an A2L parser library from scratch for OpenXCP would be out of scope. The A2L standard is mainly used by the automotive industry and therefore not well-known outside of this industry sector. ASAM decided against a XML migration in the early 2000s. The reasoned decision claims, that use of XML would increase file size and therefore slow-down parsing. Additionally existing automotive tools would need to be rewritten from scratch. [ASA17b]
OpenXCP takes this step and uses the widespread JSON data exchange format. The software architecture for serialization in JSON is described in subsection 4.2.3. An example of a OpenXCP JSON file can be found in the appendix A 'OpenXCP manual'. JSON was chosen as project-file and XCP-configuration-file, because the JSON standard is a very common format for data exchange today. JSON files are less verbose and smaller in size than XML based files. This leads to faster processing. Therefore, JSON is well suited for XCP based tools and addresses the issues ASAM mentioned regarding XML. [STR15]
The Qt frameworks offers parsing functionality for reading and writing JSON files. The JSON file is used to save project related settings and XCP measurement and calibration data. The naming and data types are adopted from the latest 'ASAM MCD-2 MC' standard [ASA15c]. This makes it easier for users to recognize keywords, which they are used from an A2L file.
Nevertheless, a project requirement is to exchange XCP measurement and calibration configuration with third-party XCP master tools. Consequently, OpenXCP offers an A2L export feature. Tools like CANape can import an A2L file generated by OpenXCP and continue the calibration work. OpenXCP uses a template A2L file to generate the final A2L file. Transport layer settings, DAQ events, measurement signals and calibration parameters are inserted into the template. This method allows to easily produce a standard conform A2L file, without the need of an A2L parser. Listing Listing 5.1 shows an excerpt of an A2L file generated by OpenXCP. It includes a Ethernet UDP/IP configuration and a measurement signal. The measurement entry, for example, contains the memory address *'0x70012474'*, symbol name *'vehicleSpeed'*, data

type *'uint64'*, physical unit *'km/h'* and minimum/maximum allowed value *'-50/300'*.

```
/* OpenXCP Ethernet */
/begin XCP_ON_UDP_IP
        0x0100
        0x15B3 /*port: 5555*/
        ADDRESS "192.168.0.128"
/end XCP_ON_UDP_IP
/* OpenXCP measurement / characteristic */
/begin MEASUREMENT vehicleSpeed "vehicleSpeed"
        A\_UINT64 NO_COMPU_METHOD 0 0 -50 300
        READ_WRITE
        ECU_ADDRESS 0x70012474
        ECU_ADDRESS_EXTENSION 0x0
        FORMAT "%.15\"
        /begin IF_DATA CANAPE_EXT
                100
                LINK_MAP "vehicleSpeed" 0x70012474 0x0 0 0x0 1 0x0 0x0
                DISPLAY 0 -50 300
        /end IF_DATA
        SYMBOL_LINK "vehicleSpeed" 0
        PHYS_UNIT "km/h"
/end MEASUREMENT
```

Listing 5.1: A2L excerpt: Ethernet/Measurement configuration

5.1.3. Record history

At the end of a measurement and calibration session, the monitored data is saved to a record file. This is a comma-separated-value (CSV) file. It is generated with the *qtcsv* library [Che17]. In the appendix A 'OpenXCP manual' an example CSV-file is included. The filename contains a time- and date-stamp. The file itself also includes the date- and time-stamp of the recording. A header for each column is present to describe the data. The first column *t[s]* lists the time in *seconds*, relative to the files timestamp. Each of the

following columns contain the value of a single signal. Every header of a signal-column is composed of a symbol name and a physical unit. The number of signal-columns represent the number of measurement-signals during recording.
A row in the record file can contain one or multiple signal-values. If a signals value was captured at the same point in time as another signal (e.g. two signals from a 100ms event), than the values have the same timestamp. Therefore, the values are written to the same row. The implementation calculates the difference between two timestamps, to see if they are equal. A epsilon value of 1 msec is used for the subtraction to reduce the number of rows and therefore the file size. This can be turned off for better accuracy or the epsilon can be increased to further reduce the number of rows. The timestamps are generated in the back-end. When the *XcpTask* receives a new value from the slave, it sends a signal to the back-end. There a Qt-timestamp method is called to generate an relative integer timestamp in ms.

5.1.4. Visual data representation

With a third-party program, like Microsoft EXCEL, MATLAB [The18] or CANape [Gmb18], the record CSV file can be imported. With these tools the data can be further evaluated. For example, calculations based on the data or a visual representation. Figure 5.1 shows an example diagram of a sinus curve. The pairs of value and time got recorded by OpenXCP from an ECU internal variable *'sinus'*. The data was saved to a record CSV file and imported to Microsoft EXCEL. Out of the data, this diagram was created.

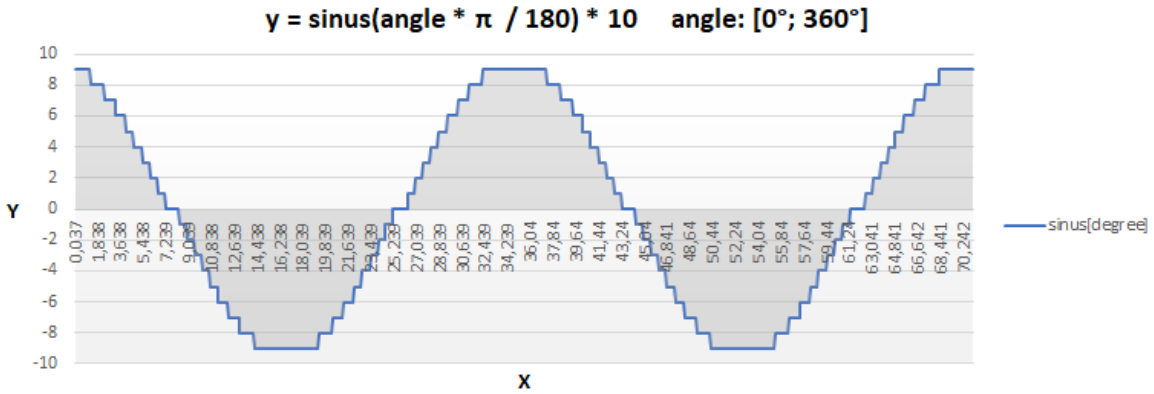


Figure 5.1.: Measurement sinus curve

5.1.5. Parsing sequence

OpenXCP uses three steps to acquire the data needed for operation. This includes parsing the ELF file for symbol names, memory address and data type. Parsing the DWARF DIE-Tree for additional debugging information. And parsing the C-source code to obtain XCP attributes from the OpenXCP comments. The software architecture is described in subsection 4.2.2.
The order of parsing these three steps affects the overall speed of data acquisition. It also affects the algorithm implementation, which used for parsing. In the current version of OpenXCP this order of parsing is used:

1. ELF parser
2. DWARF parser
3. C-source code OpenXCP comment parser

First all global variables are resolved from the ELF file. Then the DWARF tree is parsed, to resolve all complex data types, nested variables and to calculate the corresponding nested memory addresses. The DWARF tree also includes the file path for each variable. This file path information is used by the source parser to find the OpenXCP comments. A list of variables is also passed to source parser from the ELF and DWARF parser. The source parser therefore knows in which file it has to search for a matching OpenXCP comment for a specific variable.
Another option would be to use the following order of parsing:

1. C-source code OpenXCP comment parser
2. ELF parser
3. DWARF parser

First all source code files are parsed, searching for OpenXCP comments. The found variables and XCP attributes would be saved. Then the ELF file would be parsed. A list of variables from the source code parser and the ELF parser are passed to the DWARF parser. Now the DWARF parser only has to resolve the variables in the list.
The idea behind the second method is to reduce computing time and improve scalability. Recursively stepping through the DWARF tree is computationally intensive. This could

be reduced by only resolving the actual variables with a OpenXCP comment. This would especially be helpful with a project containing a very large number of overall variables, but a small number of variables for measurement and calibration.
On the other hand, the implemented method (first) resolves all variables and can therefore be used without the source parser. The attributes contained in the OpenXCP comments could also be entered manually in the GUI by the calibrator. Another benefit, is that the source comment parser only has to parse a selected number of files.
In conclusion, the best parsing order depends on the specific project.

5.1.6. C-source code comments parser

The source comment parser is able to automatically collect XCP attributes for signals and parameters. The calibrator does no longer have to manually input the XCP attributes by a GUI interface. The developer of the source code can provide these attributes in place during development. The software developer can set the attributes according to the project specification and implementation requirements. The calibrator can measure and modify the slave in the given framework. For example the lower- and upper limit of a signal/parameter is provided (@XCP_LowerLimit, @XCP_UpperLimit). The calibration value must be in between of those limits. This helps to protect against calibration settings, that harm the ECU or damage the plant [PZ16, p.73]. XCP attributes supported by OpenXCP are listed in Listing 5.2. The purpose of the individual keywords and values can be looked up in the 'ASAM MCD-2 MC' standard [ASA15c].
The OpenXCP comment is Doxygen conform. Every attribute is prefixed by *'@XCP_'* token. This prefix is not reserved by other Doxygen commands. Therefore, OpenXCP comments can be used as 'custom commands' together with Doxygen [Hee17]. For convenience of adding a OpenXCP comment to an internal variable, a comment template is provided for the HighTec/Eclipse IDE.

```
1  /**
2  * @brief XCP measurement variable: vehicle speed
3  * @XCP_Measurement
4  * @XCP_Comment Vehicle speed
5  * @XCP_LowerLimit -50
6  * @XCP_UpperLimit 300
7  * @XCP_Discrete false
8  * @XCP_MaxRefreshRate 10
9  * @XCP_PhysUnit km/h
10 * @XCP_ConversionFunction
11 * @XCP_ReadWrite ro
12 */
13 volatile int32 vehicleSpeed;
14 /**
15 * @brief XCP characteristic showcase variable: modifyMe
16 * @XCP_Characteristic
17 * @XCP_Comment showcase for XCP characteristic
18 * @XCP_LowerLimit 0
19 * @XCP_UpperLimit 255
20 * @XCP_PhysUnit
21 * @XCP_StepSize 1
22 */
23 volatile uint8 modifyMe;
```

Listing 5.2: OpenXCP source code comment

Algorithm The *ParserManager* calls the *SourceParser* with a list of variables. The *SourceParser* opens the associated source file for each variable. The file is read line by line. Each line is evaluated by the state machine in the *ProcessLine*-method shown in figure 5.2. The algorithm looks for the start of a comment /**, followed by a line starting with *@XPC* token. It is distinguished between a measurement- and a characteristic-comment, because each supports a different set of attributes. If a measurement- or

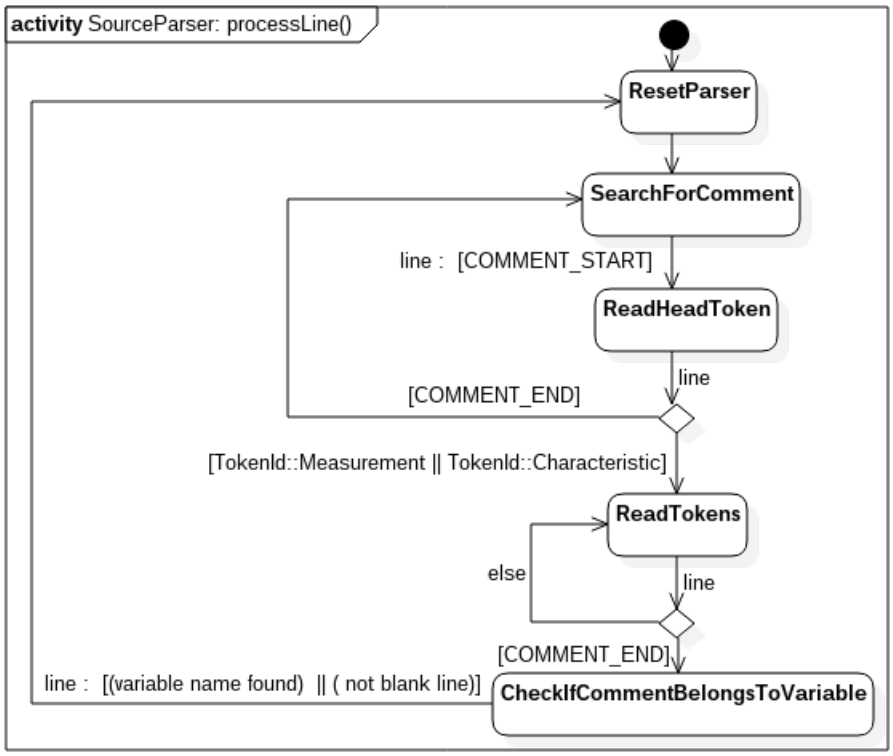


Figure 5.2.: *ProcessLine*-method state-machine,*SourceParser*

characteristic-token ('HeadToken') is found, then the following lines are scanned for attributes. At the end the closing comment line **/ is expected. If this is the case, the file is further processed until a line with a variable is found. Next, the variable from the line is compared to the variable from the 'search list'. If the variables match, than the OpenXCP comment belongs to this variable and the attributes are saved to the variables dataset. The *ProcessLine*-method state-machine is reset and the next variable in the 'search list' is processed. To avoid processing already parsed lines of a file, the lines are added to an ignore-list. This improves parsing for source files, which contain more than one signal or parameter.

5.1.7. Choice of ELF/DWARF parser library

OpenXCP heavily relies on an ELF/DWARF parser. To seamless interact with the ELF file a library is needed. The well known *GNU Binutils* provide a set of command line tools to read and manipulate object files, among other things. These tools are included in the GNU Compiler Collection for an ECU development tool chain. Even though, these tools are very helpful during firmware development, it is difficult to integrate them programmatically into OpenXCP. Calling a binary tool from OpenXCP and parsing the command line output from this tool is not a good approach. A more clean approach is to directly use a library for parsing. [Ben11]
The following open source libraries support parsing ELF/DWARF:

1. BFD libbfd (used by the GNU binutils) [Fre17]
2. libelf/libdwarf [And17]
3. libelfin [Cle17]

Good reasons for choosing *libelfin* instead of the other two options: In contrast to BFD, *libelfin* has an easy accessible documentation including working example code. BFD lacks documentation [Fre17, BFD README file]. Also *libdwarf* has a good documentation, but recently some critical vulnerabilities were reported [And17]. Some more reasons for choosing *libelfin*:

- Written in C++11, supports C++11 features
- Syntactic parsing of DWARF/ELF (not semantic)
- Supports DWARF 2,3,4
- Well designed API
- Easy to integrate in Linux based programs

[Cle17] On the other hand, *libelfin* is labeled as not production ready and the software developer still needs a good semantic understanding of DWARF to use the library. Also building the library under Windows is a bit of a hassle. These disadvantages are discussed in the chapter 'Results' under subsection 6.2.1. [Cle17]

5.1.8. ELF parser implementation

The Executable and Linkable Format (ELF) is i.a. an executable format for binary images. It can include a symbol table, sections and debugging information (DWARF). More information on how a ELF file is organized can be found in [TIS95].
The ELF parser implementation is straight-forward, due to the use of the *libelfin* library. The *elfParser* class reads the ELF file. For each ELF symbol with the type 'data object' the symbol name, memory address and memory size is saved to a data-structure named *ElfInfo*. Each *ElfInfo* object represents a variable and its attributes in the source program.

5.1.9. DWARF parser implementation

For source-level debugging the DWARF file format was introduced. It can be used with all object file formats. Commonly it used with ELF. DWARF describes a program as a tree data-structure. The tree contains nodes, which may contain children and siblings. Each node can represent a variable, a data-type or a subroutine of the source program. [Eag12], [Sri13]
The source program is represented by debug information entries (DIEs). A single DIE has an identifying tag and a list of attributes. One or more DIEs together describe an associated entity in the source program. Attributes of the DIE describe the entity in detail. They contain constants, variables or a reference to another DIE. The 'Compilation Unit DIE' is the root of all DIEs in a compilation unit (source file). It includes information, such as file path, file name and the compiler that produced the DWARF data. [Eag12], [Sri13]
Figure Figure 5.3 shows three different DIEs. A name of a tag has the prefix *DW_TAG* and an attribute name has the prefix *DW_AT*. The tag name describes the type of the DIE. Each DIE type has specific attributes. In the figure, the first DIE describes a variable, the second a member of a struct and the last one a basic data type. The *DW_TAG_variable* DIE has the name *var* and has a reference to its compilation unit and its line number in that file. Also a reference to the data type of *var* is provided. The attributes of the DIE *DW_TAG_member structElement* are similar. In this example, both DIEs have a reference to the same data type DIE. Here, the *DW_TAG_base_type* describes an integer basic data type with a size of four bytes.

```
<1> DW_TAG_variable
        DW_AT_name var
        DW_AT_decl_file 0x1
        DW_AT_decl_line 0x78
        DW_AT_type <0x3>
        DW_AT_location <exprloc>
<2> DW_TAG_member
        DW_AT_name structElement
        DW_AT_decl_file 0x1
        DW_AT_decl_line 0x50
        DW_AT_type <0x3>
        DW_AT_data_member_location <exprloc>
<3> DW_TAG_base_type
        DW_AT_byte_size 0x4
        DW_AT_encoding 0x5
        DW_AT_name int
```

Figure 5.3.: DWARF Debugging Information Entry (DIE)

OpenXCP uses the libelfin library to parse the DWARF tree. The DWARF tree is parsed in three steps.
In the first step the variables of the source program are determined/resolved and saved to a data-structure. The second step looks up the corresponding struct of the struct-elements and adds the elements to the struct. The last step calculates the individual memory addresses of the struct-members and the memory size of an array.

1. parse step Figure Figure 5.4 illustrates the first step. The parser steps through the DIE nodes of the DWARF tree for each compilation unit. The parser starts at the root node and steps further down the DWARF tree. Therefore, each node is evaluated. The children and siblings of a node are evaluated recursively. Therefore, the recursion gets deeper as the parser evaluates the children and siblings of the root node. The attributes of the DIEs/nodes obtained by this process get saved in a data structure called *dwarfTreeObj*. All *dwarfTreeObj* are stored in a map for further processing in step two and three.
The diagram shows that the source path of a compile unit, the structs and the variables are analyzed. In this context, the term 'variable' means: standard variable, array or struct-member. This type of DIEs are added to the *dwarfTreeObj* data-structure, after their attributes have been evaluated. A attribute can either be a variable, a constant or a

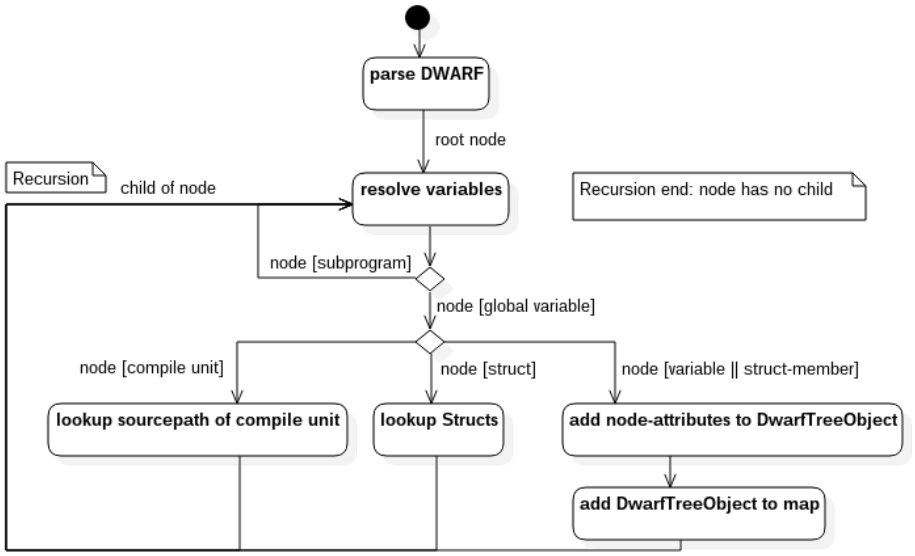


Figure 5.4.: Parse DWARF tree

reference. For the latter, the parser has to follow the reference until a constant is found. For example this is the case for user defined data types, that point to a basic data type. Analyzing the attributes is not shown in the diagram. The parser stops the recursion, if a node has no child anymore.

2. parse step After the DWARF tree was parsed and a map of *dwarfTreeObj*s was created, the parser searches for the structs that belong to the struct-members. Instead of using the original DWARF tree, this step parses the reduced dataset in the *dwarfTreeObj* map. This step also includes a recursion, because a struct can contain a nested struct. The recursion ends, if a struct only contains standard variables, arrays or struct-members. The recursion can also end, if the maximum recursion-depth was reached. This artificial exit-point is needed, if the parser can not fully evaluate the members of a struct.

3. parse step In this step the memory address of each struct member is calculated. The address of the struct and the memory size of each struct-member is known. The address

of a member is determined by adding the memory size of the previous members to the base address of the struct. Also in this step the total size of an array is determined, by combining the result of the ELF parser with this parser result.

Result As a result of the DWARF parsing process, all global standard variables and complex variables(struct, array) are determined with the corresponding data type, source path and so on. This result is saved in derived classes of *parseResultBase*.

5.1.10. CTO exchange

This section describes the how the master exchanges CTO messages with the slave.
All XCP commands are identified by an PID and have a command specific payload structure. OpenXCP stores this information, according to the XCP protocol standard, in a header file. Each XCP command is defined in an individual C++ namespace. This allows convenient access to the attributes of each command. In the appendix Table D.1 a list of supported commands by OpenXCP master is provided.
The *xcpTask* class is responsible for managing the XCP data exchange with the slave. Each command has a separate method for composing its specific XCP payload.
Following steps are undertaken to send a command to the slave: First, the payload gets assembled and the command is added to the *CommandPayload* data-structure. This data-structure is used to process the command and to match the slaves response to the command request. The *CommandPayload* is then added to a queue. The command-queue is ordered by the FIFO-principle. According to the programs control flow the commands in the queue are send to the slave. Therefore, the XCP-packet is embedded in a UDP datagram. A timeout timer is started. The master then waits for a response of the slave. The command-queue is processed sequentially. This means, that the next command can not be sent until the response of the current command was received. This complies withe XCP specification. A request has no PID and therefore can only be identified by its order. The UDP class informs the *xcpTask* class about a response from the slave. This is done by the Qt signal-slot mechanism. Then the XCP payload response is matched with the masters request. Further, the payload is processed in the according method of the specific command. Now, the next command from the command-queue can be send.

The access to the send and receive process is protected by a mutex. While this is blocked, it is still possible to add commands at the end of the command-queue. Access to the command-queue is also protected by an additional mutex. The mutex manages the read and write access.
Polling commands for measuring signals are added to the command-queue by a timer based approach. Each signal has an attached Qt-timer, that gets called according to the polling interval of the signal. If the timer is triggered, the polling command is added to the queue, including the payload for the specific signal.The timer based approach is well suited for accurate timing and is scalable. A loop, which checks each signals timing interval, would become very inaccurate and CPU intensive, for a growing number of signals.
All other commands, such as standard commands, are added to the command-queue according to the program flow and the XCP protocol standard.
The FIFO command-queue is processed sequentially by best effort.

5.1.11. Synchronous data transfer (DAQ)

OpenXCP supports synchronous data transfer with dynamic DAQ lists. Figure 5.5 shows the command sequence for a DAQ dynamic list configuration.
The XCP protocol standard defines the command sequence to correctly configure the slave for DAQ with a dynamic list allocation. The standard provides an example, including payload data, for DAQ dynamic configuration [ASA15b, p.272ff.]. This example complements the sequence diagram mentioned above.
The DAQ configuration CTO commands are exchanged with the slave as described in subsection 5.1.10 'CTO exchange'. The sequence shows the configuration for one measurement signal with one event. This is the simplest case. [ASA15b, p.272ff.] defines which commands must be repeated to measure more than one signal and several events. OpenXCP uses for-loops to add the required commands in the correct sequence to the command-queue.
For example, for measuring two signals with an identical event, the *WRITE_DAQ* command must be send two times. This command configures an ODT entry of the DAQ list. It includes the ECU memory address of one signal. Also the number of ODT entries must be configured to match the number of signals (e.g. two).

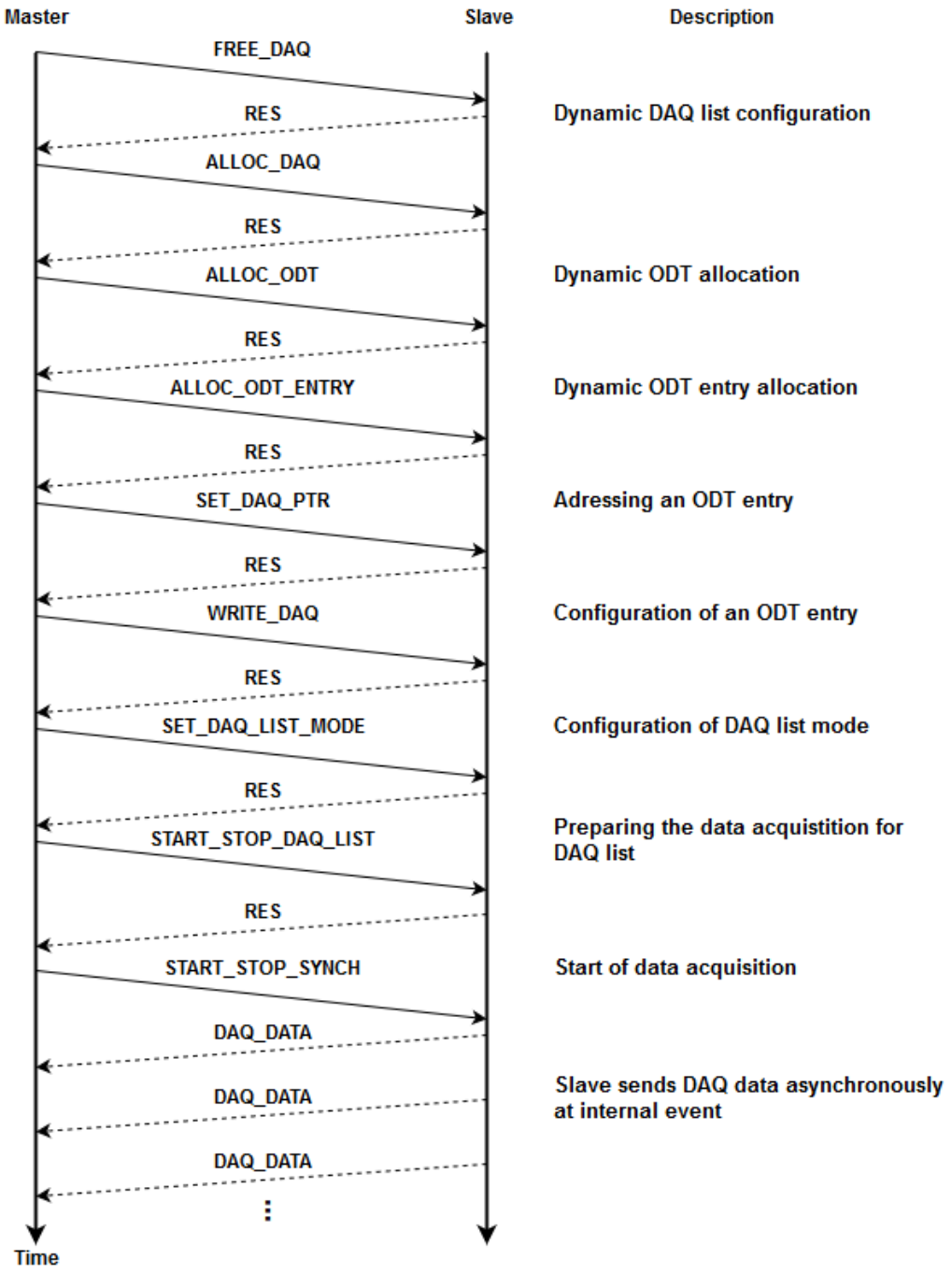


Figure 5.5.: DAQ dynamic configuration sequence

Each event needs one individual DAQ list. The first event has the number 0 and the following events are continuous [Tri05, p.15].
For more than one event the DAQ configuration must specify the number of DAQ-lists, the number of ODTs for each DAQ-list/event and the number of ODT entries (signals) per ODT. Again, each signal must be configured with its memory address (ODT entry). At the end of the configuration, for each DAQ-list/event the *DAQ-mode* must be configured and each DAQ-list/event must be marked as selected. With the command *START_STOP_SYNCH* the data acquisition is started for all selected DAQ-lists.
The slave starts to send DTO-DAQ messages to the master asynchronously, every time an internal ECU event is triggered (event synchronous). The DTO messages are received asynchronously by the master. A DTO message does not contain a PID. OpenXCP uses the 'absolute ODT' packet addressing, as described in section 3.6 'DTO packet addressing'.
One DTO message can contain data from one or more signals from one event. In order to match the payload data to the correct signal, OpenXCP uses following method: After the response was forwarded to the *responseProcessDaq* method the DTO payload gets evaluated. OpenXCP keeps a list of the configured DAQ-lists with the corresponding ODT and ODT entries. Therefore, it is known which DAQ lists contains which signals. The position of each signal in a ODT (DTO payload) is determined by the offset. The offset is calculated by the size of the data type from a signal. In a loop all ODT entries are processed one after another.
In this version OpenXCP only supports one ODT per DAQ-list. As one ODT can have a large number (configurable) of ODT-entries this has not proven to be a big disadvantage.

5.1.12. Deploy OpenXCP

The OpenXCP master software is deployed on Windows 7/10 by following these steps:

1. Comment out the debug-mode in the qmake project file
2. Run qmake
3. Build OpenXCP with 'release' settings with MinGW 32bit
4. Copy the OpenXCP execution file to a new folder for deployment
5. Add Qt dependencies to this folder by running *windeployqt*
6. Find missing third-party dependencies with *dependencywalker*
7. Check if all libraries/DLLs are included according to Table C.1

5.2. Slave

The subsection 4.3.1 describes the slaves architecture model from a abstract position. Complementary this section describes the slaves implementation by a file based approach. Table 5.1 subdivides the slave by layers and list the according files to each layer.

Layer	Files	Description
Protocol layer	xcp_cfg.h	Configuration
	xcp_def.h	Default settings
	xcp_par.h/.c	Parameter
	xcpBasic.h/.c	Implementation
Transport layer	udpXcp.h/.c	UDP implementation
Network layer	ipStackAurix.h/.c	Network (lwIP stack)
Application	typedefs.h	Type definionts for XCP driver
	xcpTask.h/.c	XCP init / event loop
	xcpTargetPlattform.c	Callback functions for ECU specific tasks

Table 5.1.: Files of the slaves XCP driver

The core of the protocol layer is the implementation, which puts the XCP protocol standard into code. The driver and the XCP settings are configured by preprocessor directives in the *xcp_cfg.h* header file. The XCP settings must match the XCP settings in the OpenXCP master. Otherwise no working XCP session can be established. The parameter file and the default settings file do not have to be changed. The standard settings are fine.
The transport driver implements the Ethernet UDP/IP transport layer in this setup. The implementation for this is located in the *udpXcp* files. An additional transport layer, for example USB, can easily be added here.
The network layer includes a file for initializing the lwIP stack and functions for polling received Ethernet packets.
The *application* includes files that manage the complete XCP driver and integrate it into the existing business logic of the ECU. The type definition file bridges the gap between ECU and XCP driver data type definition. The *xcpTargetPlattform* is used by the protocol driver to call ECU, transport layer and application specific functions. In this file the ECU specific callback functions are registered, which implement the stub functions from the protocol driver. The *xcpTask* file includes functions to initialize the complete XCP driver. It is also the gateway to the business logic of the ECU. The main-loop calls the 'event-loop' function in this file to trigger DAQ events.

5.2.1. Parameters and signals in RAM

Several different calibration concepts exist to modify parameters. This section will briefly introduce some common calibration concepts. OpenXCP supports calibration in RAM, which is explained in detail.

Flash calibration ECUs ready to go into production normally store the global parameters value in flash memory. The following C code *const int par = 5;* defines that the variable 'par' is located in the *.rodata* section of the flash memory. The flash memory can not be calibrated by XCP during runtime (online). Therefore, the parameter must either be modified offline in the source or in the HEX file. The first option requires to compile, link and flash the complete firmware to take effect. The second option requires to modify the parameters value in the HEX file and to flash the file again. [PZ16, p.80 f.]

RAM calibration A improved concept is to calibrate parameters in RAM. This is refereed as online calibration. During development all parameters are located in RAM to easily modify their values. If the project reaches production state, no further testing and calibrating is needed and the parameters can be moved back to the flash memory. [PZ16, p.82]
The following C code *volatile int par = 5;* defines the variable 'par' in RAM. The initial value '5' is stored in flash (*.text*) and the variable is initialized during ECU startup by the startup code. The initialized global/static variables are located in the *.data* section. Uninitialized global/static variables are located in the *.bss* section. It is important to use the 'volatile' keyword. This informs the compiler, that the variable can be modified externally and therefore no memory optimization should be executed. [PZ16, p.82]
OpenXCP supports this online calibration method to measure signals and calibrate parameters in RAM.
Further to mention is, that after a reboot of the ECU the calibration data is lost and the parameters are initialized again from the original value in the flash. To recover the state of the last calibration session, all values for the parameters have to retransmitted via XCP. This is discussed in section 6.3.

Advanced calibration methods XCP provides advanced calibration techniques that are currently not supported by OpenXCP. This for example includes 'flash overlay' concept. This method has the benefit, that a set of parameters can be changed at once. Therefore the plant is always in a stable state. This method is described in section 3.7 'Memory page switching'. Further information to 'flash overlay' and other advanced calibration concepts can be found in [PZ16, p.84 ff.].

5.2.2. Toolchain

Following tool-chain was used for developing software on the Infineon Aurix TriCore:

- HighTec Free TriCore Entry Tool Chain, version 4.6 and 4.9
- GCC 4.6 and GCC 4.9 (TriCore GCC)
- AURIX TC277x TFT Application Kit Firmware SW V1.1
- HighTec Development Platform Version: 1.6 (Eclipse based IDE)
- Device Access Server (DAS) v6.0.0

The tool-chain version used with the Application Kit Firmware can be changed by setting the path to the tool-chain in the *'CfgCompiler_Gnuc'* makefile. The GCC version determines the standard DWARF version of the ELF file. As the DWARF version has impact on parsing the DWARF tree, the information above should be noted. subsection 5.1.9 explains this fact in more detail.

5.2.3. Lightweight IP stack

Attribute	**raw/callback API**	**sequential API**	**socket API**
OS needed	no	yes	yes
Portable code	**no**	**no**	**yes**
Require threading	no	yes	yes
Callback mechanism	**yes**	**no**	**no**
Performance	high	low	low
Memory usage	**low**	**high**	**high**

Table 5.2.: Compare lwIP APIs

The basis of the XCP transport Ethernet driver is the lwIP stack. The open source network stack is included in the 'AURIX TC277x TFT Application Kit Firmware SW V1.1' and is especially developed for embedded systems. It offers three different APIs: 'raw/callback' API, high-level 'sequential' API and high level 'socket' API. [Dun17]
The following table shows the attributes of the APIs compared to each other: Next, attributes of the table are described and it is explained why the 'callback' API is most

suitable for the XCP transport driver in this project.
The project definition specifies to develop an XCP driver on a bare metal ECU without a RTOS. Therefore, only the 'callback' API can be used, because the other two APIs require an OS. Both high level APIs use threading provided by an OS. One thread handles the application code, which uses the API, and another thread runs the lwIP stack. 'The model of execution is based on the blocking open-read-write-close paradigm' [Dun17]. Hence, the network throughput performance is lower and the memory footprint higher compared to the 'callback' API. This is because of the higher abstraction level and the multi-threaded paradigm. On the other hand, the 'callback' API works event-based in a non-blocking single-threaded environment. Application code and network stack are executed in the same thread, which avoids switching threads and the overhead of the thread-context. This API uses 'callback' functions to inform the application about received network packets. The 'callback' API and the 'sequential' API have the drawback, that the source code is not compatible with other networks stacks. The source code is lwIP specific. The 'socket' API is compatible with other POSIX OSes. As mentioned above, the 'callback' API was chosen, because it is the only API that works in a non OS-environment. [Dun17]

Implementation The lwIP stack is used in the 'mainloop mode'. The lwIP stack is only called from the main-loop and never from an ISR. This mode uses the consumer-producer paradigm. The producer is the Ethernet IRQ, that puts the received Ethernet packets in a queue. A function in the main-loop consumes this queue, by polling the queue for received packets. [Dun17]
Section subsection 5.2.4 explains how the lwIP stack is used to receive and send XCP packets over UDP.

5.2.4. XCP over UDP

Initialization The lwIP stack is initialized, with the 'callback' API, before entering the main-loop. This includes setting the MAC- and IP address. Next, the network stack is configured for a UDP connection. Therefore, a new UDP protocol control block (PCB) is allocated in memory. The PCB is bound to a local IP address and port. Next, a callback-function is registered that will be executed, if a UDP datagram is received for

the PCB. This setup, uses a unconnected PCB, which means that data can be send to any specified remote IP address and receive datagrams from any remote IP address. This is necessary, because the slave does not know the masters IP address and port during setup. Therefore, a connected PCB can not be used. The masters IP and port is set after the first XCP over UDP packet was received from the master. [Dun17]

Receive In the main-loop the lwIP queue is polled for received Ethernet packets. The registered 'receive-function' is called. In this function, the remote IP address and port is saved and the UDP payload is mapped on a data-structure, that represents the XCP-packet structure. The XCP payload is forwarded to the XCP-protocol-driver function: *XcpCommand*. This function evaluates the XCP packet and interprets the XCP command [Tri05, p.35].

Send The XCP-protocol driver calls the *ApplXcpSend* function to transmit a XCP packet to the master. The ECU and transport layer specific implementation of this stub function is the *udpXcpSend* function. This function expects a XCP payload and the size of the payload as parameter. The function adds the Ethernet specific XCP header to the XCP payload and copies both header and and payload in a XCP-Ethernet-frame data structure. For each XCP packet that is send, a packet counter is incremented, according to the standard [ASA15a].
DTO-DAQ Several XCP packets can be sent in one UDP datagram, in the case of DTO-DAQ. Therefore, the XCP-Ethernet-frame is added to a *sendBuffer*. The XCP packet is not send immediately. Each time a new XCP packet is sent, the *udpXcpSend* function is called and a new XCP-Ethernet-frame is added to the *sendBuffer*. To determine whether the *sendBuffer* has reached its capacity, a global counter is incremented with the size of each XCP-Ethernet-frame. If the value of the counter has reached the size of the *sendBuffer*, then the UDP packet, including several XCP-Ethernet-frames, is sent to the slave. To accomplish this, the *ApplXcpSendFlush* stub function is called, which calls the transport layer specific udpXcpSendFlush function. This function actually copies the complete *sendBuffer* to the lwIP stack packet buffer (PBUF). Next, the UDP datagram is sent to the master by calling the send-function of the lwIP stack. Finally, the *sendBuffer*, the counters and the PBUF are reset.

CTO In case of standard CTO commands and polling the *sendBuffer* is not used and the XCP-Ethernet-frames are send in a single UDP datagram each. This is necessary, because the slave needs to response to a masters request individually. Hence, the protocol driver calls the *ApplXcpSendFlush* function to 'flush' and send each XCP packet separately. For example for command response packets (RES) and error packets (ERR).

5.3. Open-source license and publication

5.3.1. OpenXCP master

OpenXCP is licensed under GPLv3 and published on GitHub:
`https://github.com/shreaker/OpenXCP`.
The license terms can be summarized as follows: 'You may copy, distribute and modify the software as long as you track changes/dates in source files. Any modifications to or software including (via compiler) GPL-licensed code must also be made available under the GPL along with build & install instructions'[Wan13]. A copy of the license is included in OpenXCP. OpenXCP uses the following software and resource, which are compatible to GPLv3:

Software	Description	License
QT[1]	QT framework	i.a. GPLv3
libelfin[2]	**ELF/DWARF parser**	**MIT**
qtcsv[3]	CSV library	MIT
Icons[4]	**Gnome icon theme 3.12**	**GPLv2**

Table 5.3.: Software/resource used by OpenXCP (license)

[1] `https://www1.qt.io/licensing-comparison/`
[2] `https://github.com/aclements/libelfin`
[3] `https://github.com/iamantony/qtcsv`
[4] `http://ftp.gnome.org`

5.3.2. Slave

For this project an Infineon TriCore ECU was used as a XCP-slave. The base software 'AURIX TC277x TFT Application Kit Firmware SW V1.1' is property of Infineon AG. The XCP driver for the protocol layer is property of Vector Informatik. The company offers the XCP basic version 'freely via the internet' [Tri05]. Both components are not distributed with OpenXCP. The XCP transport layer driver for Ethernet UDP/IP was developed during this project and therefore is published with OpenXCP under GPLv3. This XCP Ethernet driver is compatible with the XCP basic driver from Vector Informatik. For integration of the complete XCP driver it is advised to obtain the XCP basic driver directly from Vector Informatik and the XCP Ethernet driver from the OpenXCP GitHub repository.

6. Result

This chapter concludes the success of the project by presenting three evaluations. The first test explains how measurement quality and performance differs between polling and DAQ mode. The second test shows the fault tolerance of OpenXCP master and slave. The last test evaluates the accuracy of packet timing in polling and DAQ mode. The next section discusses problems encountered during development and offers solutions.
The last section highlights future goals of the OpenXCP project.

6.1. OpenXCP performance test

DAQ compared to polling This section describes the result of a performance test. The goal is to analyze the performance of OpenXCP master and a Infineon Aurix ECU slave in DAQ and polling mode. The aim is to find out how much CPU power and bandwidth is used by OpenXCP master and slave during polling and DAQ mode. These two measurement methods are compared to each other. Also the measurement quality and the maximum number of measurement signals is evaluated.
Test environment:

- Master: Intel Core i7-6700K CPU @ 4.00GHz, Windows 10
- Slave: Infineon AURIX Application Kit TC277 TFT, CPU @ 200Mhz, 472 KB RAM
- Benchmark master: Resource Monitor (Windows 10)
- Benchmark slave: Performance measurement algorithm included in AURIX TC277x TFT Application Kit Firmware SW V1.1 (see Listing 6.1)
- 100 Mbit/s Ethernet UDP/IP connection

- Measurement signal data type: *uint16*
- Polling interval: every 10ms
- DAQ intervall: 1 event with 10ms
- DAQ memory size 1024 byte (kXcpDaqMemSize)
- DTO packet size: 80 byte
- DAQ-DTO-packets are cumulated in an UDP-packet

In Listing 6.1 a snippet of the slaves performance measurement algorithm is shown. It calculates the CPU load. The algorithm is divided in a *main-while-loop* and an ISR context. It uses the 'CPU Clock Count Register CCNT' to count the number of CPU clock cycles. In the 'main-loop' a 'cpu idle counter' is incremented, while the CPU is not busy executing some ISR. The more the CPU is under load, the slower the 'cpu idle counter' is incremented. The difference between the last 'cpu idle counter' value and the actual 'cpu idle counter' is calculated in the 'ISR_PREF' This ISR is called periodically. Line 11 in the listing shows the formula, which is used to actually calculate the CPU load.

```
1  while(true){ //main while loop
2  cpu_CCNT_actual = CPU_CCNT;  //CPU Clock Count Register
3  cpu_CCNT_diff = cpu_CCNT_actual - cpu_CCNT_last;
4  cpu_CCNT_diff_min = min(cpu_CCNT_diff_min, cpu_CCNT_diff);
5  cpu_CCNT_last = cpu_CCNT_actual;
6  cpu_idle_counter++;    //Idle-counter for cpu load measurement
7  }
8  ISR_PREF(){ //Interrupt service routine
9  counter_diff = cpu_idle_counter - cpu_last_count_value;
10 cpu_last_count_value = cpu_idle_counter;
11 cpu_load = 100 - counter_diff / (cpu_freq / 100 / cpu_CCNT_diff_min);
12 }
```

Listing 6.1: Aurix performance CPU load algorithm snippet

In Table 6.1 the result of the benchmark-test is shown. The 'No measurement' row

Mode	Signals	Slave CPU	OpenXCP CPU[1]	OpenXCP TX	OpenXCP RX	Note
No measurement	0	2%	0%	0 Kbit/s	0 Kbit/s	XCP idle
Polling	**6**	**3,9%**	**72%**	**6,7 Kbit/s**	**3,9 Kbit/s**	
DAQ	6	2,3%	32%	0 Kbit/s	1,8 Kbit/s	
Polling	**40**	**4,1%**	**80%**	**7,8 KBit/s**	**4,5 KBit/s**	**Loss of measuring points**
DAQ	40	2,5%	88%	0 KBit/s	8,5 KBit/s	

Table 6.1.: Benchmark OpenXCP polling/DAQ: 10ms

is the reference for the test. The slave CPU has a basic workload of 2% due to the operations of the firmware. This workload must be subtracted from the results in 'Slave CPU' column to obtain the additional CPU-workload by DAQ and polling.
In conclusion, polling mode is more CPU intensive for the slave as well as for the master. Also the traffic load on the transport layer is higher. This is due to the fact, that for polling, two messages must be exchanged for each signal. Especially the load of the masters request (TX) is high. This is because the request includes 8 bytes XCP-payload plus 4 bytes of XCP-Ethernet-header. The response (RX) of the slave uses less bandwidth, because it includes 2 bytes XCP-payload (uint16 signal) and 2 bytes of XCP-Ethernet-header. Of course the XCP-packets are embedded in an UDP/IP packet, which must be added to the network utilization. For synchronous data exchange (DAQ) no request from the master is needed. Therefore, the master sends no data at all to the slave. Also the network utilization of the received packets from the slave is less. This is because, the slave adds several DTO-XCP-packets into one UDP packet. This reduces the overhead.
During the performance test, it was revealed, that the polling mode does not produce reliable measurement results for more than six signals measured simultaneously at an interval of 10ms. Polling 40 signals, showed that the master could not handle the packet load. Measuring points were lost and the timing of the measurements was inaccurate. For DAQ mode the maximum reliable number of simultaneously measured signals, at an interval of 10ms, is 40. More measurement signals fully utilize the masters CPU, which results in malfunction.

[1]OpenXCP uses one CPU-core. Value refers to a single-core. Intel i7-6700k provides four cores.

Fault tolerance The XCP connection between master and slave is fault tolerant. If the Ethernet connection is interrupted measurement points are lost. As soon as the Ethernet connection is established again, OpenXCP automatically continues the XCP record session.

Timing This test evaluates the packet timing accuracy. Polling and DAQ measurement mode are compared on how accurate the timing of the measurement is.
The timing of DAQ and polling mode was measured with *RawCap*. The packet trace was analyzed with *Wireshark*. The test environment was identical to the one listed under section 6.1. Wireshark can calculate the time delta between two network packets. In the best-case, every XCP response from the slave, is received by the master at exactly the 'interval-time' (set-point). Following scenarios were tested for monitoring six measurement signals simultaneously:

Scenario	interval / setpoint	max. deviation	average deviation
Polling	10 ms	10 ms	5 ms
Polling	**100 ms**	**20 ms**	**9 ms**
DAQ	10 ms event	1 ms	0 ms
DAQ	**100 ms event**	**1 ms**	**0 ms**

Table 6.2.: Timing: DAQ / polling

The result in Table 6.2 shows that DAQ mode is reliable and accurate. On the other hand, polling can not be recommend, if timing accuracy is of great importance.

6.2. Discussion

The following section discusses the ELF/DWARF parsing process. The last section explains the benefits of verifying the slaves memory integrity.

6.2.1. ELF/DWARF parser

During the development of the DWARF parsing functionality of OpenXCP a lot was learned about the DWARF file format. This knowledge can be used to further improve

the DWARF parsing implementation.
As mentioned *libelfin* is not production ready yet. Therefore, it may be beneficial to change the DWARF library. Also *libelfin* is focused on Linux, which made it difficult to compile the library under Windows.
Besides that, parsing the ELF file was not possible anymore after updating the slaves toolchain from GCC 4.6 to GCC 4.9. The reason for this is that the C-compiler version 4.6 creates an ELF with DWARF version 2 and the C-Compiler version 4.9 an ELF file with DWARF version 3. The parser has to distinguish between the DWARF versions, because the attributes/values of the DIEs vary. The OpenXCP DWARF parser was eventually successfully tested with DWARF version 2 and 3. More details on the DWARF versions can be found in [Eag12, p.2f.].
Following command options should be used to produce a compatible ELF file:
gcc main.c -g -gdwarf-3 -O1
The option *-g* tells the compiler to include debug information in the object file (e.g. ELF). With *-gdwarf-3* the DWARF version 3 is selected. This is useful, if the compilers default DWARF version is not the desired version. The optimization switch *-O* defines which optimization level the compiler should use. The best level for debugging is level *0*. For the Aurix slave ECU the minimum level is 1, otherwise the firmware would not function correctly. This is because of the lacking optimization, which increases memory usage and CPU usage. Still level 1 is good for debugging.
The idea to parse the MAP file produced by the linker, instead of the ELF file, proved to be misleading. The MAP file is a human-readable ASCII file format, which helps the developer to evaluate the memory layout of the firmware. The MAP file generation is optional and each compiler produces a different layout. Therefore, the standardized binary ELF file is obviously better suited for parsing.

6.2.2. Memory integrity

The XCP measurement and calibration process is based on reading and modifying data from ECU internal memory addresses. Therefore, it is essential, that the XCP masters memory information about the slaves memory layout match exactly. The slaves XCP driver does not verify the memory address, which was send by the master over a XCP command. The master is responsible to always provide the correct memory address,

otherwise the slave can malfunction, as well as the connected plant can be damaged.
To verify the slaves memory integrity, XCP offers a *BUILD_CHECKSUM* command. Hereby, the memory integrity can be verified after flashing the ECU and it can be checked, if the masters ELF file matches the ECU memory.
In the current version OpenXCP only supports the comparison between the checksum of the ELF file saved in the JSON project file and the checksum of the current ELF file selected by the user for parsing.
Also in OpenXCP the XCP command *SET_MTA* and *BUILD_CHECKSUM* are implemented. The first command sets a pointer to the memory address from which the *BUILD_CHECKSUM* starts calculating a checksum over a selected memory region. The slave has successfully been configured to support *ADD* and *CRC16-CITT* checksum algorithms.
In the next step the OpenXCP master must calculate the checksum over a given memory region from the Intel-HEX file. This file exactly represents the memory data from the slave. For the time being, the HEX-file could only be evaluated manually. The checksum for a number of variables was successfully calculated and then compared to the received checksum calculation of the slave. In the next release of OpenXCP it is planned to implement a Intel-HEX parser for this task and to add a *CRC16-CITT* algorithm implementation for calculating the checksum.
Also it is important to mention, that it is necessary to calculate the checksum over the *.rodata/.text* section as well as the *.data* section. The first checksum is used to verify, if the binary is identical. The second is used to verify, if the initialization of the ECUs variables was successfully executed by the startup code.
The following command creates an Intel-HEX file from a given ELF file:
objcopy -O ihex slave.elf slave.hex

6.3. Future work

The following list highlights the upcoming work on OpenXCP. Including improvements on existing functionality and new features.

Recover calibration values after ECU reboot After reboot the ECU variables are initialized from the data in the flash memory. The OpenXCP master must therefore retransmit the values from the last calibration session. Therefore, the slaves memory integrity must be compared to the masters settings. After that, the saved values in the JSON project file could be retransmitted.

Improve DWARF parsing As explained in subsection 5.1.5 the DWARF parsing computing time could be reduced by preselecting the relevant variables. This can be done by parsing the source files for relevant XCP variables first. Also the implementation could be improved: subsection 6.2.1.

Support floating-point numbers *Double* and *float* variables should be supported by OpenXCP master.

Fully support *BUILD_CHECKSUM* As discussed in subsection 6.2.2, it is important to verify the slaves memory integrity to avoid malfunction due to accessing invalid or wrong memory addresses.

Add multi-threading to OpenXCP Implement a GUI- and a XCP-thread to increase the number of simultaneously measured signals while keeping the GUI responsive.

Port OpenXCP to Linux as this would introduce OpenXCP to a new user group. Building OpenXCP for Linux is well-prepared, because Qt/C++ and the libraries in use are well suited for Linux.

Port XCP driver to an ARM ECU with USB as transport layer The widespread use of ARM ECUs makes this an reasonable goal. The XCP protocol driver needs no adjustment. Only the XCP on USB driver must be implemented.

Look for contributers to the OpenXCP project to achieve the future goals mentioned above.

List of Figures

List of Tables

Bibliography

[A15] K. A. *Highest measurement and calibration performance through cooperation of OEM, Tier1, uC, Tool-Supplier*. Springer Vieweg, Wiesbaden, 2015.

[And17] D. Anderson. *David A's DWARF Page*. 2017. URL: https://www.prevanders.net/dwarf.html (visited on 01/09/2018).

[ASA15a] ASAM. *ASAM MCD-1 (XCP on Ethernet), Universal Measurement and Calibration Protocol, Ethernet Transport Layer*. ASAM e.V., 2015.

[ASA15b] ASAM. *ASAM MCD-1 (XCP), Universal Measurement and Calibration Protocol, Protocol Layer Specification*. ASAM e.V., 2015.

[ASA15c] ASAM. *ASAM MCD-2 MC (ASAP2 / A2L), Data Model for ECU Measurement and Calibration*. ASAM e.V., 2015.

[ASA17a] ASAM. *ASAM MCD-1 XCP*. 2017. URL: https://www.asam.net/standards/detail/mcd-1-xcp/ (visited on 12/20/2017).

[ASA17b] ASAM. *ASAM MCD-2 MC*. 2017. URL: https://www.asam.net/standards/detail/mcd-2-mc/wiki/ (visited on 01/05/2018).

[AUT17] AUTOSAR. *Specification of Module XCP*. 2017. URL: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_XCP.pdf (visited on 01/03/2018).

[Ben11] E. Bendersky. *How debuggers work: Part 3 - Debugging information*. 2011. URL: https://eli.thegreenplace.net/2011/02/07/how-debuggers-work-part-3-debugging-information (visited on 01/15/2018).

[Che17] A. Cherepanov. *Library for reading and writing csv-files in Qt*. 2017. URL: https://github.com/iamantony/qtcsv (visited on 01/05/2018).

[Cle17] A. Clements. *Libelfin, C++11 ELF/DWARF parser*. 2017. URL: https://github.com/aclements/libelfin (visited on 01/07/2018).

[CLG11] P. Caliebe, C. Lauer, and R. German. "Flexible integration testing of automotive ECUs by combining AUTOSAR and XCP". In: *2011 IEEE International Conference on Computer Applications and Industrial Electronics (ICCAIE)*. Dec. 2011, pp. 67–72.

[D17] M. D. *Entwicklung einer generischen Testumgebung für Automotive Software Systems*. 2017. URL: http://www.qucosa.de/fileadmin/data/qucosa/documents/21858/Masterarbeit_Daniel_Markert.pdf (visited on 12/31/2017).

[Dun17] A. Dunkels. *Lightweight IP stack 2.0.2*. 2017. URL: http://www.nongnu.org/lwip/2_0_x/raw_api.html (visited on 01/11/2018).

[Eag12] M. Eager. *Introduction to the DWARF Debugging Format*. 2012. URL: http://www.dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf (visited on 01/15/2018).

[Ele17] Elektrobit. *EB tresos AutoCore*. 2017. URL: https://www.elektrobit.com/products/ecu/eb-tresos/autocore/ (visited on 12/31/2017).

[Fre17] I. Free Software Foundation. *GNU Binutils*. 2017. URL: https://www.gnu.org/software/binutils/ (visited on 01/11/2018).

[Gmb17] V. GmbH. *Microsar Product Information*. 2017. URL: https://vector.com/portal/medien/cmc/info/MICROSAR_ProductInformation_EN.pdf (visited on 12/31/2017).

[Gmb18] V. I. GmbH. *CANape*. 2018. URL: https://vector.com/vi_canape_en.html (visited on 01/15/2018).

[He 09] S. X. He Y. *An XCP Based Distributed Calibration System*. 2009.

[Hee17] D. van Heesch. *Doxygen*. 2017. URL: https://www.stack.nl/~dimitri/doxygen/index.html (visited on 01/07/2018).

[Kle12] A. Kless. *High-speed measurements for electric and hybrid vehicles*. 2012. URL: https://vector.com/portal/medien/cmc/press/PMC/E-Mobility_VX1131_HanserAutomotive_201205_PressArticle_EN.pdf (visited on 12/20/2017).

[LP16] G. Lazar and R. Penea. *Mastering Qt 5*. Packt Publishing, Limited, 2016.

[Ltd17] T. Q. C. Ltd. *Signals & Slots*. 2017. URL: https://doc.qt.io/qt-5/signalsandslots.html (visited on 12/06/2017).

[MA17] P. Markl and S. Albrecht. *Analyzing AUTOSAR ECU software with XCP*. 2017. URL: https://vector.com/portal/medien/cmc/press/Vector/AUTOSAR_Monitoring_HanserAutomotive_SH_201111_PressArticle_EN.pdf (visited on 12/20/2017).

[MPC08] M. Muresan, D. Pitica, and G. Chindris. "Calibration parameters principles for MATLAB S-functions using CANApe". In: *2008 31st International Spring Seminar on Electronics Technology*. May 2008, pp. 105–110.

[Nor11] O. I. de Normalización. *ISO 26262: Road Vehicles : Functional Safety*. ISO, 2011.

[Onl14] C. N. Online. *Testing and debugging embedded software*. 2014. URL: https://can-newsletter.org/software/software-miscellaneous/140522_microsar_vector (visited on 12/31/2017).

[PZ16] A. Patzer and R. Zaiser. *XCP - The Standard Protocol for ECU Development*. Vector Informatik GmbH, 2016.

[Sch14] H. H. Schwager M. *Testing and debugging of ECUs using MICROSAR AMD and CANoe.AMD*. 2014. URL: https://vector.com/portal/medien/cmc/events/Webinars/2014/Vector_Webinar_MICROSAR_CANoe_AMD_20141203_EN.pdf (visited on 12/31/2017).

[Sch94] W. Schütz. *Fundamental issues in testing distributed real-time systems*. 1994. URL: https://doi.org/10.1007/BF01088802 (visited on 12/20/2017).

[Som07] I. Sommerville. *Software Engineering*. Pearson Studium - IT. Pearson Deutschland, 2007.

[Sri13] R. Srinivasaraghavan. *Exploring the DWARF debug format information*. 2013. URL: https://www.ibm.com/developerworks/aix/library/au-dwarf-debug-format/au-dwarf-debug-format-pdf.pdf (visited on 01/15/2018).

[STR15] T. STRASSNER. *XML vs JSON*. 2015. URL: http://www.cs.tufts.edu/comp/150IDS/final_papers/tstras01.1/FinalReport/FinalReport.html (visited on 01/03/2018).

[The18] I. The MathWorks. *MATLAB and Simulink*. 2018. URL: https://www.mathworks.com/ (visited on 01/15/2018).

[TIS95] T. I. S. (TIS). *Executable and Linking Format (ELF) Specification, version 1.2*. 1995. URL: `http://refspecs.linuxbase.org/elf/elf.pdf` (visited on 01/15/2018).

[Tri05] F. Triem. *XCP Protocol Layer, Technical Reference*. 2005.

[Wan13] K. Wang. *GNU General Public License v3 (GPL-3), Quick Summary*. 2013. URL: `https://tldrlegal.com/license/gnu-general-public-license-v3-(gpl-3)` (visited on 01/13/2018).

Appendices

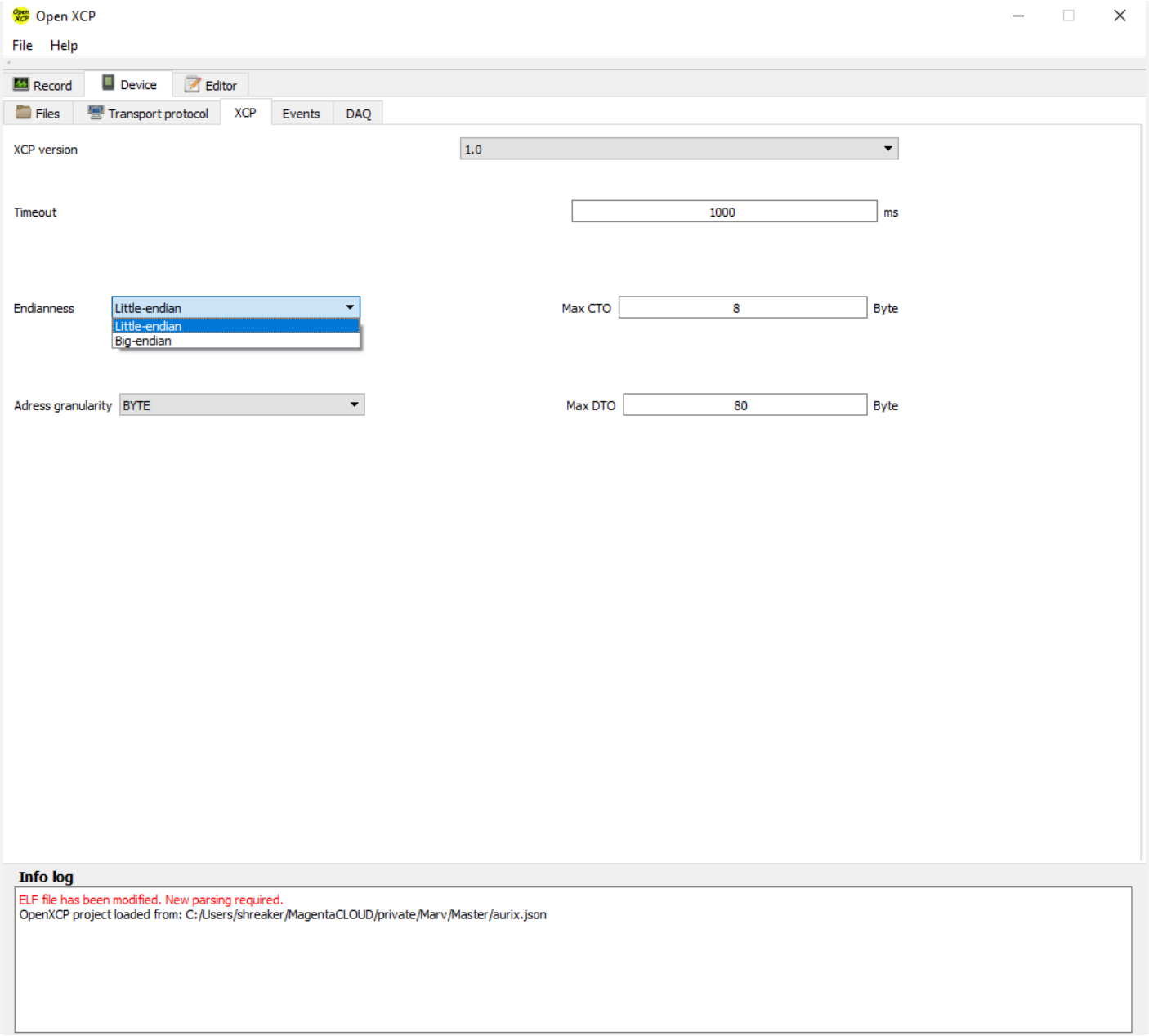
A. OpenXCP manual

OpenXCP manual

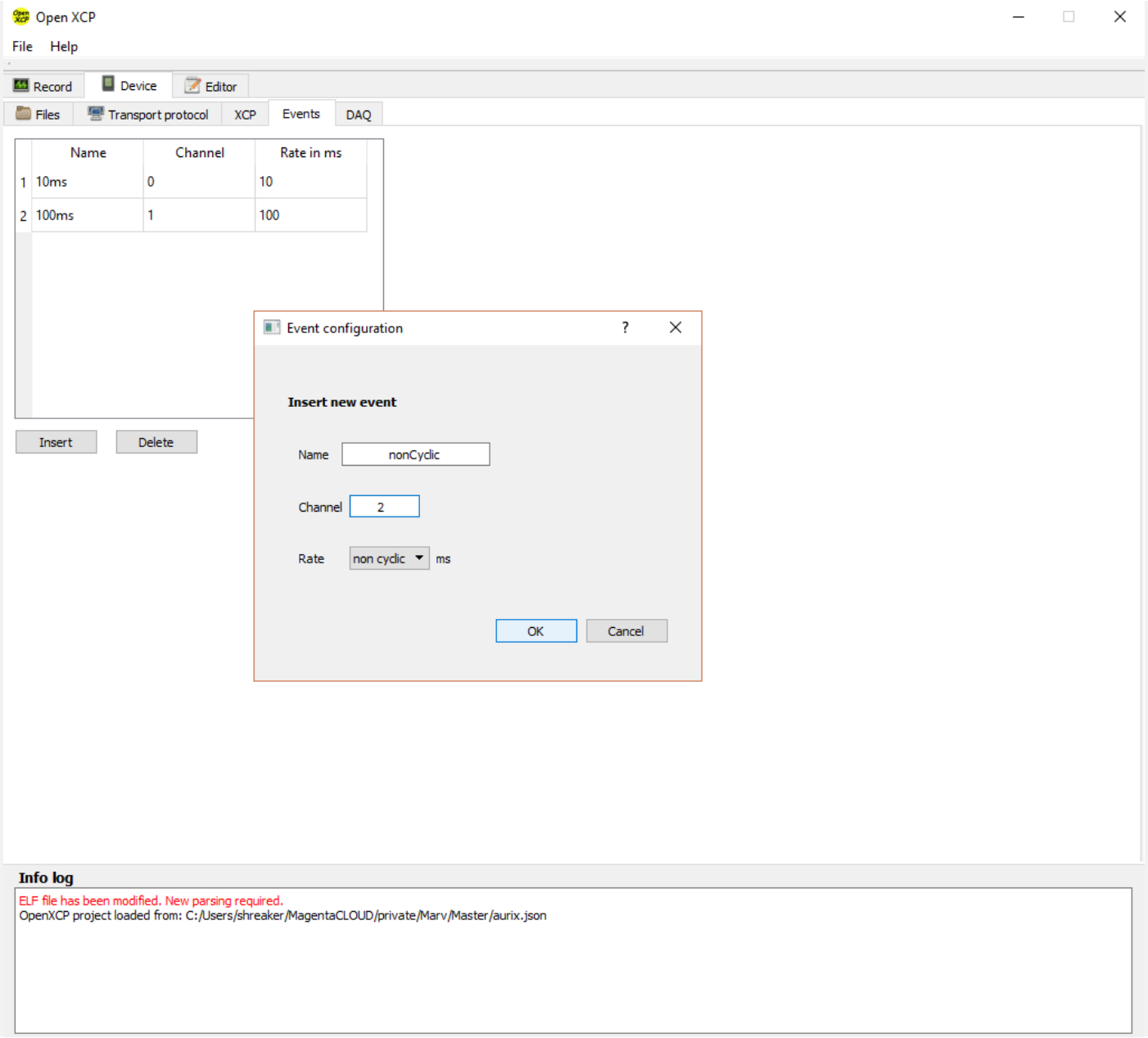
Date: 08.12.2017

Author: Michael Wolf

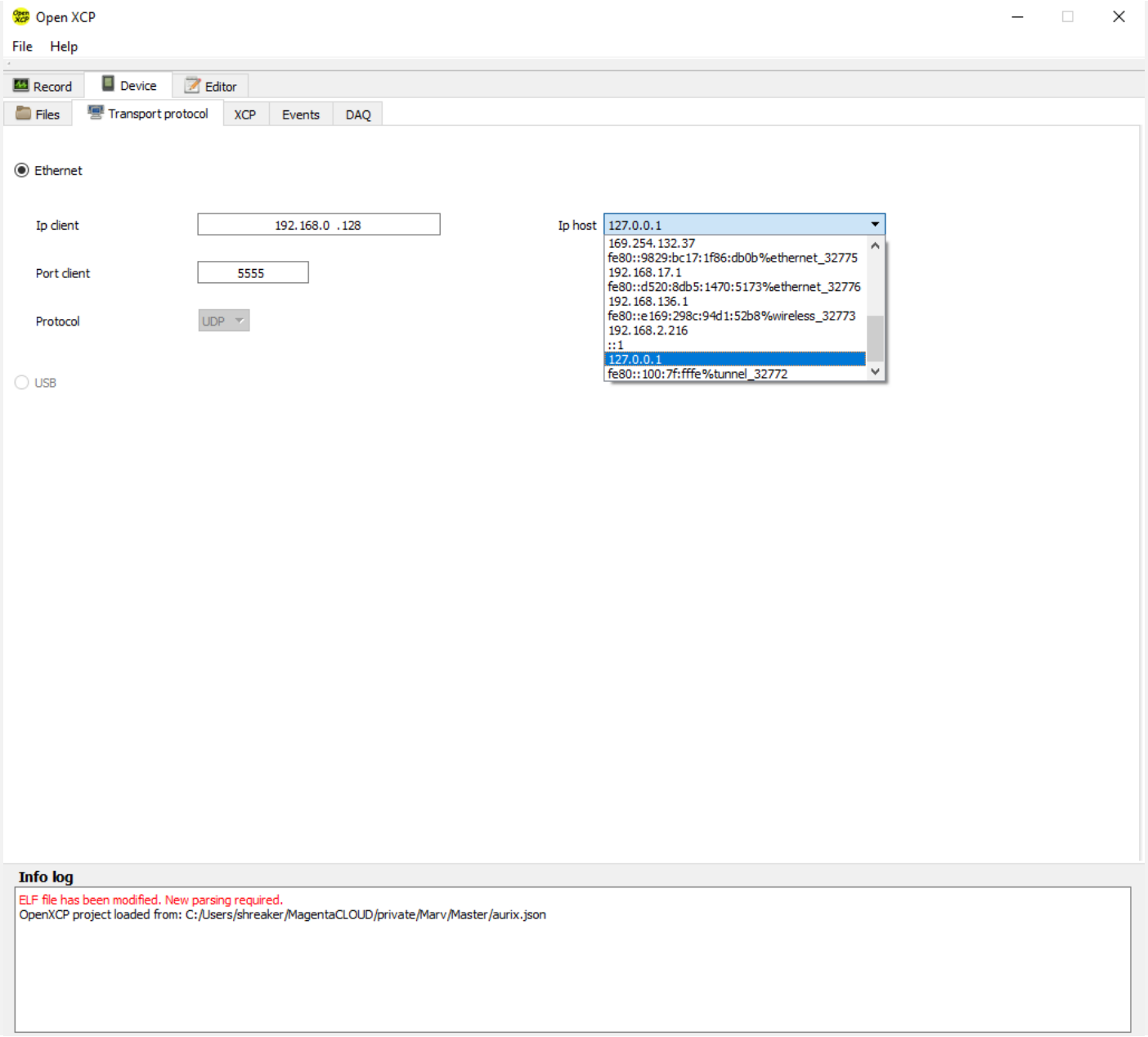
XCP configuration



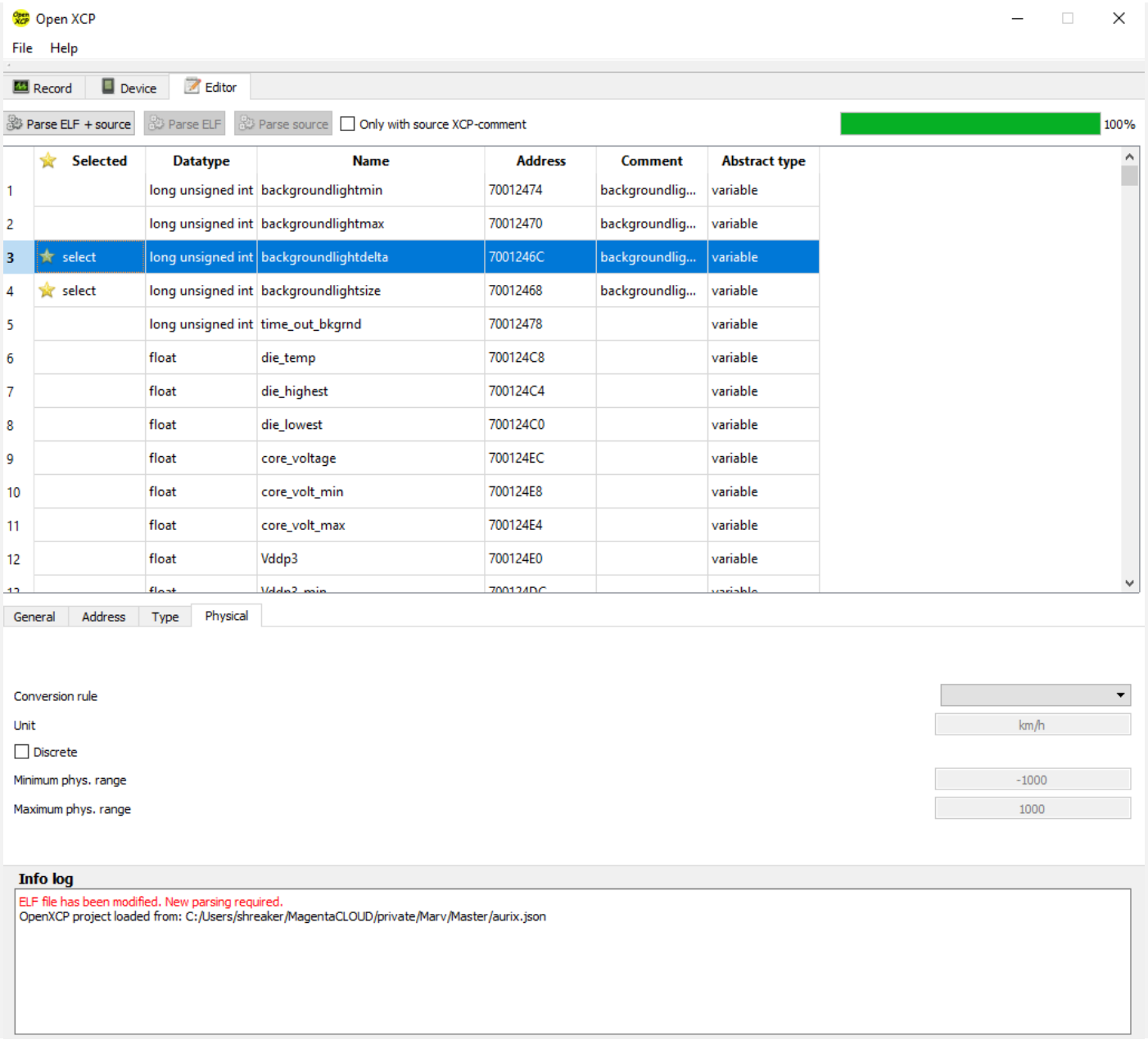
Event configuration



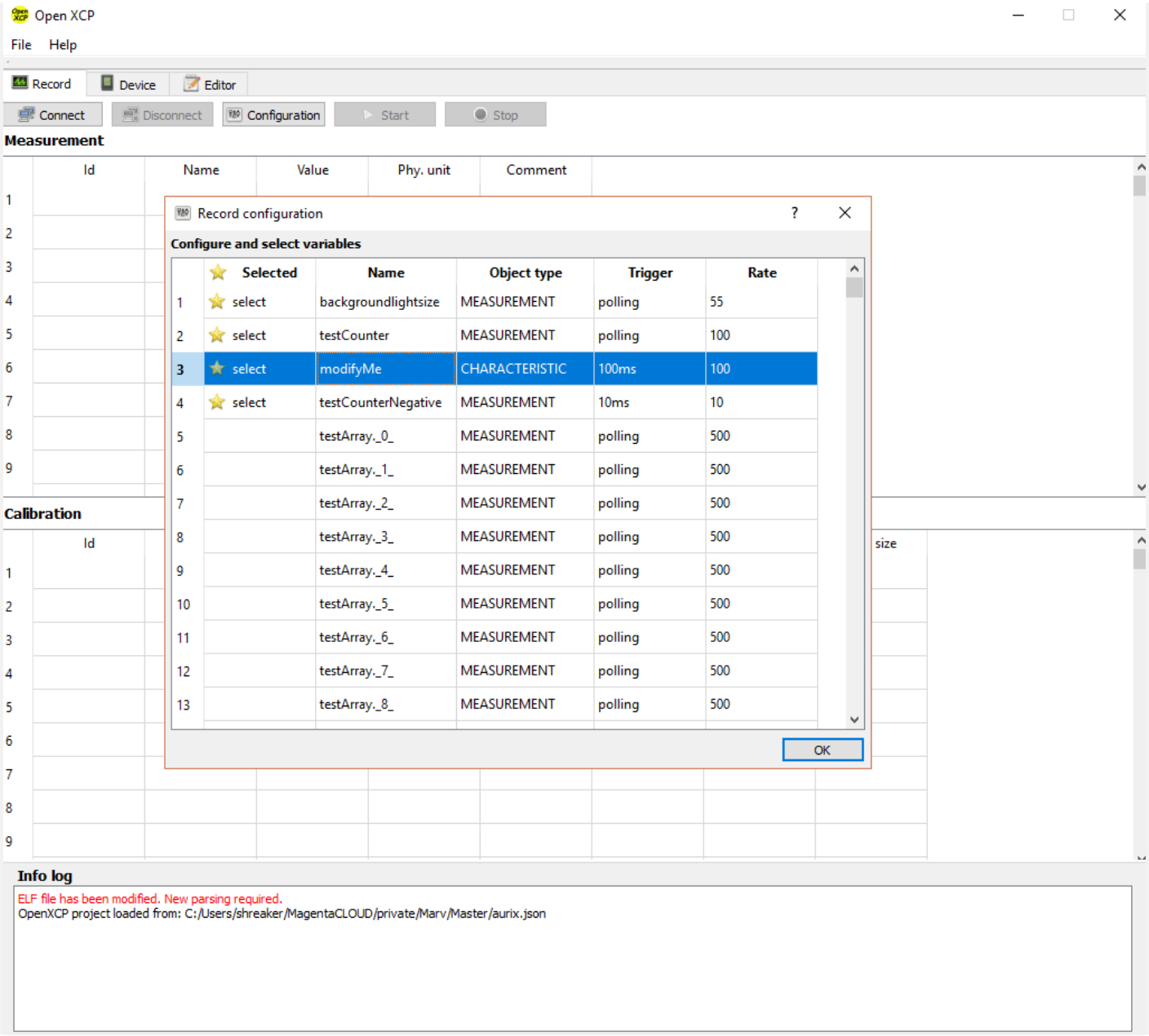
Transport protocol configurarion



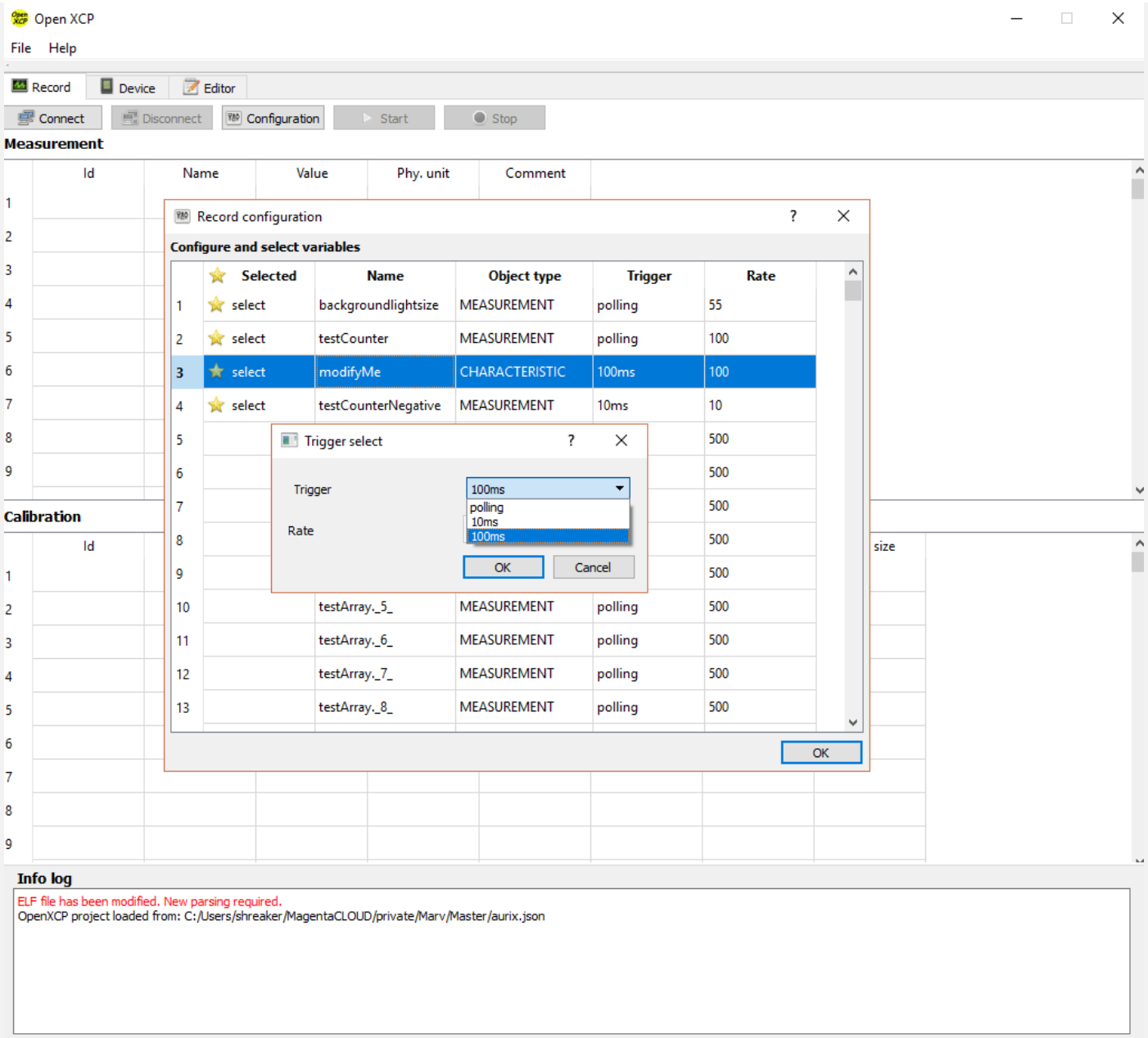
Parse ELF + source



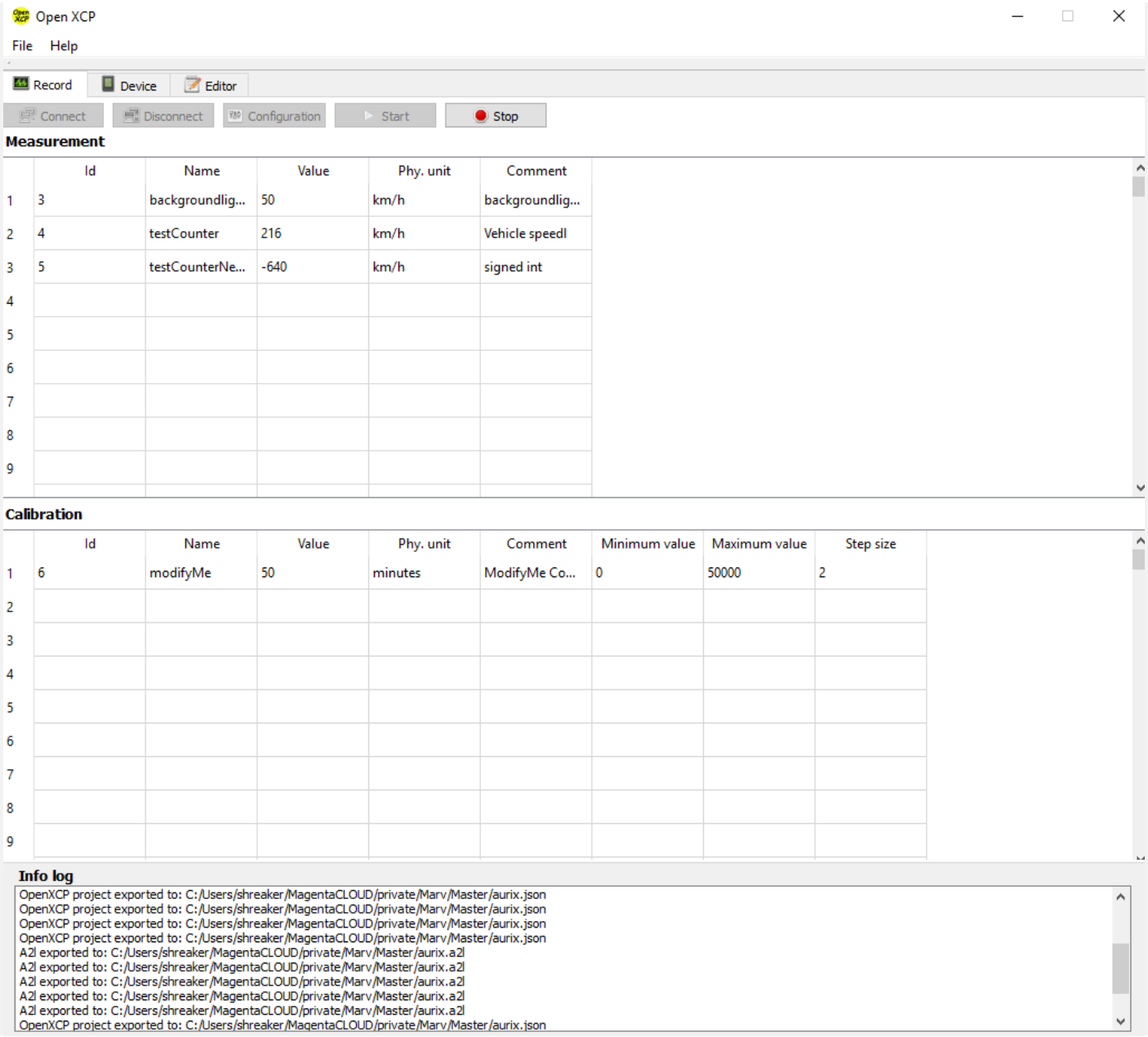
Record configuration



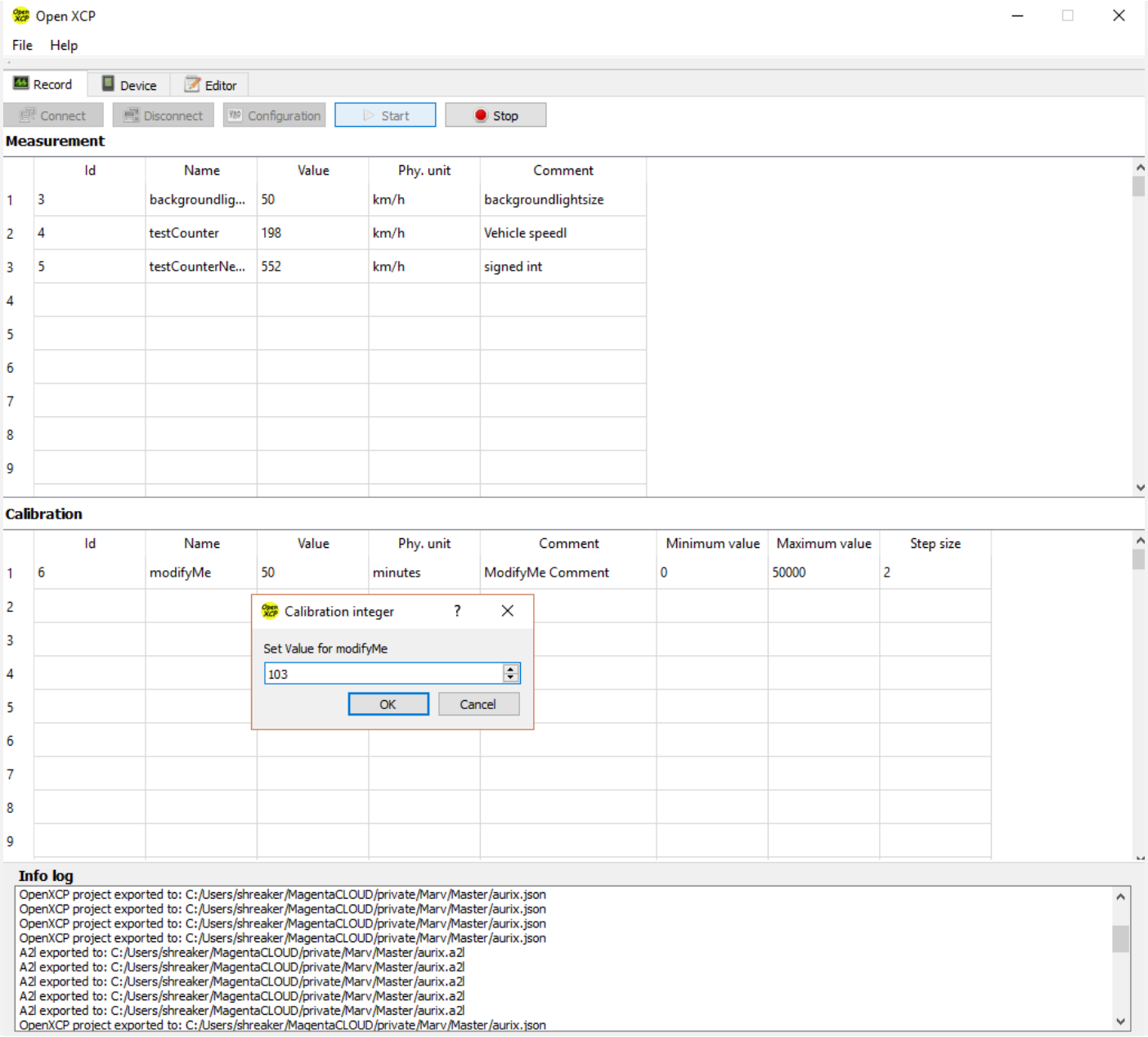
Record configuration



Record session



Record session calibrate



Record CSV example

	A	B	C	D	E
1	date	time			
2	08.12.2017	11:40:44			
3					
4	t[s]	modifyMe[minutes]	testCounterNegative[km/h]	testCounter[km/h]	backgroundlightsize[km/h]
5	0.020000		-850		
6	0.030000		-850		
7	0.039000		-850		
8	0.049000		-850		
9	0.059000		-850		
10	0.064000				50
11	0.069000		-850		
12	0.079000		-850		
13	0.089000	103	-850		
14	0.099000		-850		
15	0.104000			62	
16	0.109000		-850		
17	0.120000		-849		
18	0.125000				50
19	0.130000		-849		
20	0.139000		-849		
21	0.149000		-849		
22	0.159000		-849		
23	0.169000		-849		
24	0.179000		-849		
25	0.188000				50
26	0.189000		-849		
27	0.190000	103			
28	0.207000		-849		
29	0.217000		-849		
30	0.225000			62	
31	0.226000		-848		
32	0.234000		-848		
33	0.241000		-848		
34	0.249000				50
35	0.256000		-848		

A2L export example

A2L_Example.a2l

```
841
842     /* OpenXCP Ethernet */
843
844     /begin XCP_ON_UDP_IP
845         0x0100
846         0x15B3
847         ADDRESS "192.168.0.128"
848     /end XCP_ON_UDP_IP
849
850     /end IF_DATA
851     /begin IF_DATA CANAPE_ADDRESS_UPDATE
852     /end IF_DATA
853
854     /begin MOD_PAR ""
855     /end MOD_PAR
856
857     /* OpenXCP MEASUREMENT / CHARACTERISTIC */
858
859     /begin MEASUREMENT backgroundlightmin "backgroundlightmin"
860         A_UINT64 NO_COMPU_METHOD 0 0 -1000 1000
861         READ_WRITE
862         ECU_ADDRESS 0x70012474
863         ECU_ADDRESS_EXTENSION 0x0
864         FORMAT "%.15"
865         /begin IF_DATA CANAPE_EXT
866             100
867             LINK_MAP "backgroundlightmin" 0x70012474 0x0 0 0x0 1 0x0 0x0
868             DISPLAY 0 -1000 1000
869         /end IF_DATA
870         SYMBOL_LINK "backgroundlightmin" 0
871         PHYS_UNIT "km/h"
872     /end MEASUREMENT
```

Project file example (JSON)

B. State machine of *XcpTask* class

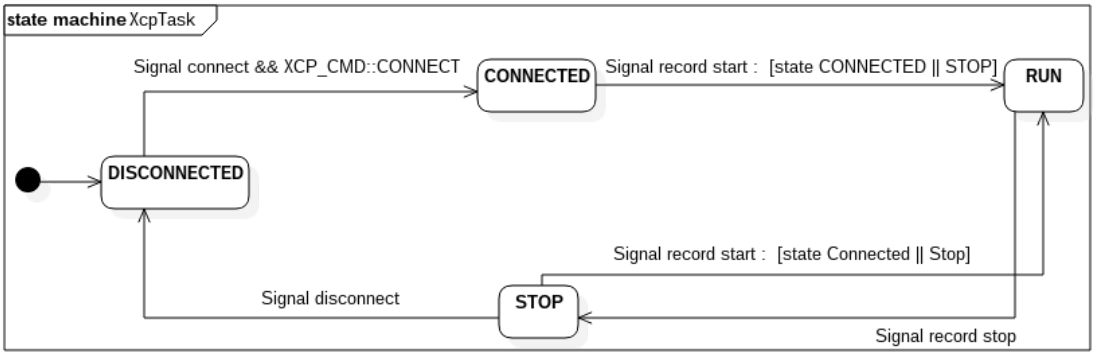


Figure B.1.: State machine of XcpTask class

C. OpenXCP dependencies

Folder	DLL name
./	D3Dcompiler_47
	libEGL
	libgcc_s_dw2-1
	libGLESV2
	libstdc++-6
	libwinpthread-1
	opengl32sw
	Qt5Core
	Qt5Cored
	Qt5Gui
	Qt5Network
	Qt5Svg
	Qt5Widgets
	qtcsv
./bearer/	qgenericbearer
	qnativewifibearer
./inconengines/	qsvgicon
./imageformats/	qgif
	qicns
	qico
	qjpeg
	qsvg
	qtga
	qtiff
	qwbmp
	qwebp
./plattforms/	qwindows

Table C.1.: OpenXCP master dependencies

D. OpenXCP supported commands

Command	PID (hex)
CONNECT	0xFF
DISCONNECT	0xFE
GET_STATUS	**0xFD**
GET_SYNC	0xFC
SHORT_UPLOAD	**0xF4**
SET_MTA	0xF6
DOWNLOAD	**0xF0**
BUILD_CHECKSUM[1]	0xF3
FREE_DAQ	**0xD6**
ALLOC_DAQ	0xD5
ALLOC_ODT	**0xD4**
ALLOC_ODT_ENTRY	0xD3
SET_DAQ_PTR	**0xE2**
WRITE_DAQ	0xE1
SET_DAQ_LIST_MODE	**0xE0**
START_STOP_DAQ_LIST	0xDE
START_STOP_SYNCH	**0xDD**

Table D.1.: By OpenXCP master supported commands

[1] Work in progress.