TECHNISCHE UNIVERSITÄT
CHEMNITZ

Development of a Calibration Tool using CCP and XCP on CAN Bus

Master Thesis

Submitted in Fulfilment of the
Requirements for the Academic Degree
M.Sc.

Dept. of Computer Science
Chair of Computer Engineering

Submitted by: Satyesh Shivam
Student ID: 386595
Date: 14.11.2016

Supervising tutor: Prof. Dr. W. Hardt
External Supervisor: Dipl-Ing Billand Dirk (Bosch Engineering Gmbh)

Abstract

Testing of the Electronic Control Unit is a very important step before delivering the software to customer. Due to increasing complexity in the requirements and the timing constraints, it is needed that the testing should be proper and on time. To meet the timing constraints it is needed to automate the entire process of testing. Although the current testing tool support the automation process, it is very slow. In this thesis a new tool has been developed which will support calibration using CCP and XCP on CAN bus. Secondly, the tool will also provide the feature of automation, where user can write their own script to test the ECU. This will make the entire process of testing very fast. Finally both the solutions will be compared with respect to time for deducing the final conclusion.

Keywords: CCP, XCP, ECU Testing, Calibration, Design Patterns

Acknowledgements

I would like to show my greatest appreciation to Prof.Dr.Wolfram Hardt for giving me an opportunity to do my thesis under his guidance.

I am grateful to Mr. Dirk Billand for being my supervisor and providing constant support in developing my thesis from all the directions. I appreciate the feedback offered by Mr. Tobias Peitsch, which was very useful in making the design better. I would like to express my deepest appreciation to Mr. Eike Bimczok for not only entrusting me with such an opportunity but also for giving constant support and valuable suggestions in development activities.

I would also like to thank Mr.Frank Ullmann, for his constant guidance throughout the thesis.

Finally, I thank my parents for their constant moral support which gave me the strength to face the challenges and succeed.

Contents

Abstract i

Acknowledgements ii

Contents iii

List of Figures vi

List of Tables viii

Abbreviations ix

1 Introduction 1

1.1 Motivation . . . 1

1.2 Research Questions . . . 2

1.3 Overview . . . 2

1.4 Organization of thesis . . . 4

2 Research Approach and Setup 5

2.1 Calibration Protocols . . . 5

2.2 Calibration Devices . . . 6

2.3 Communication Overview . . . 9

2.4 Weighted Sum Approach . . . 9

2.4.1 Protocol Selection . . . 10

2.4.2 Device Selection . . . 10

3 Protocol 12

3.1 CAN Calibration Protocol . . . 12

3.1.1 CCP Basics . . . 12

3.1.1.1 CRO Message . . . 14

3.1.1.2 Data Transmission Object . . . 15

3.1.1.3 CCP Communication 17
3.1.2 CCP Commands 18
3.1.3 Calibration using CCP 20
3.2 Universal Calibration Protocol (XCP) 22
3.2.1 XCP Basics 22
3.2.1.1 XCP Protocol Layer 23
3.2.1.2 XCP Transport Layer 25
XCP on CAN: 25
3.2.1.3 XCP Communication 26
3.2.2 XCP Commands 27
3.2.3 Calibration Using XCP: 30

4 State of the Art **32**
4.1 INCA 32
4.1.1 Introduction 32
4.1.1.1 Problem with INCA 34
4.2 CCP Measurement Tool: 34
4.2.1 Introduction: 34
4.2.2 Problems with the tool: 35

5 Architecture **36**
5.1 Design Patterns 36
5.1.1 Creational Pattern: 36
5.1.1.1 Single point of Connection 37
5.1.1.2 Processing Multiple ECUs simultaneously 37
5.1.1.3 Extension to different Transport Layer 38
5.1.2 Behavioral Pattern: 39
5.1.2.1 Modular Architecture 39
5.1.2.2 Simple Architecture 40
5.1.2.3 Tracking Activities of User 41
5.1.3 Structural Pattern: 43
5.1.3.1 Exposure of logic to third party 43
5.1.3.2 Extension to different Hardware 44
5.2 Architectural Style: 46
5.2.1 Easy to load User Interface 46
5.2.2 Handling Events 47
5.3 A2L File Parser 49

5.4 Tool Block Diagram 49
5.5 API Strategy: 50
5.5.1 Need for API Strategy: 50
5.5.2 Solution: 51
5.6 SOLID Principles: 53

6 Implementation **54**
6.1 A2L File Parser 54
6.1.1 Regular Expressions 54
6.1.2 ANTLR 55
6.1.2.1 Grammar File 55
6.1.2.2 XCP Parser Class Diagram 57
6.2 User Interface 57
6.2.1 UI Class Diagram 58
6.2.2 Validating User Interface 58
6.3 Architecture of the Tool 59
6.3.1 Overall Class Diagram 60
6.3.2 Validating Architecture 61
6.4 Validation 61
6.4.1 Calibration using GUI 61
6.4.1.1 Single ECU 61
6.4.1.2 Multiple ECU 65
6.4.2 Integration 67

7 Future Work **69**
7.1 Protocol 69
7.2 Functionality 69
7.3 Hardware 69

8 Conclusion **70**

A A2L Calibration Variable Information **71**

B A2L XCP Information **74**

C Integration Test **78**

D XCP Commands Response **80**

Bibliography **86**

List of Figures

1.1 Block diagram of testing multiple ECUs on different protocols e 2
1.2 Overall Block Diagram 3

2.1 CANcaseXL Device 7
2.2 ES581.4 Device 7
2.3 ES600 Hub 8
2.4 μLC Test System 8
2.5 Master-Slave Communication 9

3.1 Master-Slave Device Configuration 13
3.2 Command Receive Object 14
3.3 Structure of the CRO message 14
3.4 Data Transmission Object 15
3.5 Command Return Message 16
3.6 Event Message 16
3.7 Structure of CRM and Event Message 17
3.8 Message Object Communication 18
3.9 Initial Parameter setting in RAM 20
3.10 Set of commands sent for Calibration using CCP 21
3.11 Subdivision of the XCP protocol into protocol layer and transport layer . 22
3.12 XCP Frame 23
3.13 Overview of XCP Packet Identifier 24
3.14 XCP Message for CAN 25
3.15 XCP Communication Model 26
3.16 Set of commands sent for Calibration using XCP 31

4.1 INCA Variable Selection Window 33
4.2 Calibrating Variables using INCA 33

5.1 Singleton APIConnector Class 37

5.2 Multiton Pattern Example 38
5.3 Abstract Factory Pattern Example 39
5.4 Template Method Pattern Example 40
5.5 Iterator Pattern Example 41
5.6 State Diagram 42
5.7 State Machine Class Diagram 43
5.8 Facade Pattern Example 44
5.9 Adapter Pattern 45
5.10 Dependency Inversion in Hardware Devices 45
5.11 MVC Design Pattern 47
5.12 MVC Outline Diagram 47
5.13 Callback Channel Class Diagram 48
5.14 Block Diagram of the tool 50
5.15 Multi-Threaded API Access diagram 51
5.16 Multiple Network Class Diagram 52

6.1 Protocol Layer Information in A2l File 55
6.2 Grammar to extract Protocol Layer Information from A2L 56
6.3 TOKENS for Grammar 56
6.4 XCP parser Class Diagram 57
6.5 User Interface Class Diagram 58
6.6 User Interface of the Tool 59
6.7 Overall Class Diagram 60
6.8 Add A2l File in FastEcuAccess Tool 62
6.9 Progress bar for reading A2l File 62
6.10 To get the Calibration Variables of ECU 63
6.11 List of Calibration Variables in the BMW ECU 64
6.12 Selected Calibration Variables added into Main Window 64
6.13 Value to be written for calibration variables 64
6.14 Updated Values of Calibration Variables 65
6.15 List of Calibration Variables from ECU1 66
6.16 List of Calibration Variables from ECU2 66
6.17 Selected Variables from ECU1 and ECU2 67
6.18 Updated values of Calibration Variables from ECU1 and ECU2 67
6.19 Integration Test Result using INCA 68
6.20 Integration Test Result using ECUAccess 68

List of Tables

2.1 Calibration Protocol Selection 10
2.2 Calibration Device Selection 11

3.1 List of Acronyms used in XCP Communication 24
3.2 XCP Connect Command 27
3.3 XCP GET_STATUS Command 27
3.4 XCP UPLOAD Command 28
3.5 XCP DOWNLOAD Command 28
3.6 XCP SET_MTA Command 29
3.7 XCP GET_CAL_PAGE Command 29
3.8 XCP SET_CAL_PAGE Command 30

D.1 XCP Connect Command Positive Response 80
D.2 XCP Connect Command Negative Response 81
D.3 XCP GET_STATUS Command Positive Response 81
D.4 XCP GET_STATUS Command Negative Response 81
D.5 XCP UPLOAD Command Positive Response 82
D.6 XCP UPLOAD Command Negative Response 82
D.7 XCP DOWNLOAD Command Positive Response 82
D.8 XCP DOWNLOAD Command Negative Response 83
D.9 XCP SET_MTA Command Positive Response 83
D.10 XCP SET_MTA Command Negative Response 83
D.11 XCP GET_CAL_PAGE Command Positive Response 84
D.12 XCP GET_CAL_PAGE Command Negative Response 84
D.13 XCP SET_CAL_PAGE Command Positive Response 84
D.14 XCP SET_CAL_PAGE Command Negative Response 85

Abbreviations

ECU	**E**ngine **C**ontrol **U**nit
CCP	**C**AN **C**alibration **P**rotocol
XCP	Universal **C**alibration **P**rotocol
RQ	**R**esearch **Q**uestion
μLC	Micro **L**ow **C**ost Test System
ms	**m**illi **s**econd
CAN	**C**ontroller **A**rea **N**etwork
LIN	**L**ocal **I**nterconnect **N**etwork
OEM	**O**riginal **E**quipment **M**anufacturer
API	**A**pplication **P**rogramming **I**nterface
ETK	**E**mulator **T**ast**K**opf
USB	**U**niversal **S**erial **B**us
CRO	**C**ommand **R**eceive **O**bject
ODT	**O**bject **D**escription **T**able
ASAP	**A**rbeitskreis zur **S**tandardisierung von **Ap**plikationssystemen
DTO	**D**ata **T**ransmission **O**bject
CRM	**C**ommand **R**eturn **M**essage
DAQ	**D**ata **A**cquisition
UI	**U**ser **I**nterface
regex	**Reg**ular **ex**pression
MTA	**M**emory **T**ransfer **A**ddress
MVC	**M**odel **V**iew **C**ontroller
DAC	**D**igital **A**nalog **C**onverter
RTR	**R**emote **T**ransmission **R**equest

Chapter 1

Introduction

This chapter will introduce about the problem description and motivation of this thesis. Software architecture plays an important role while designing any new tool. In fact, performance of the tool is mainly dependent upon the architecture of the tool. Better the architecture, better the performance. This thesis is mainly focused on designing the architecture of tool which should have following characteristics:

- Easy to extend
- Easy to maintain
- Easy to reuse

1.1 Motivation

The final ECU firmware consists of integrated software modules. Before delivering ECU to the customer its thorough testing must be done. Testing comprises of measuring and calibrating the internal parameters of the ECU. Calibrating a signal means tuning a signal to achieve the desired performance. So, there is a need of an interface which can serve the purpose. Secondly, it might be possible that the user wants to do simultaneous calibration of multiple ECUs having different calibration protocols, as shown in the figure 1.1. Finally, the entire testing process is very time consuming and requires much more human effort, if not automated properly. So, there is a need of solution which can be integrated in an automated setup. CCP and XCP are two different calibration protocols. XCP can support different transport layers such as CAN, Ethernet and Flexray. But, CCP can only support CAN bus for communication. This thesis is focused on doing calibration using CCP and XCP on CAN bus. The detailed description of CCP and XCP has been mentioned in chapter 3 of the report.

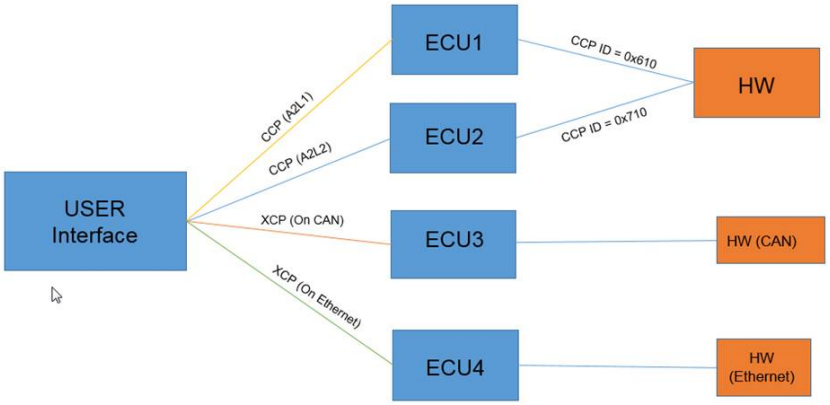


FIGURE 1.1: Block diagram of testing multiple ECUs on different protocols

1.2 Research Questions

Based on the motivation, following research questions have been derived.These questions must be answered to build a deliverable system.

- **RQ1** How the CCP and XCP protocol related information from A2L file can be read?[1]
- **RQ2** How the current design can be extended to different transport layers?
- **RQ3** How the current design can be extended to different calibration devices?
- **RQ4** How to make the system simultaneously manage multiple clients using different protocols and do calibration in parallel?

1.3 Overview

Before moving into the next chapters, it is important to put together the little pieces of findings and understand the overall system as shown in the Figure 1.2.

User Interface:
The user Interface can communicate with the API manager and display the results.

[1]File associated with ECU which gives information about the ECU's supported protocols

Tools:

This application is not limited only for test automation system but has the scope of being used by many other tools. It is the responsibility of the API Manager block to take care of these requests as shown in Figure1.1.

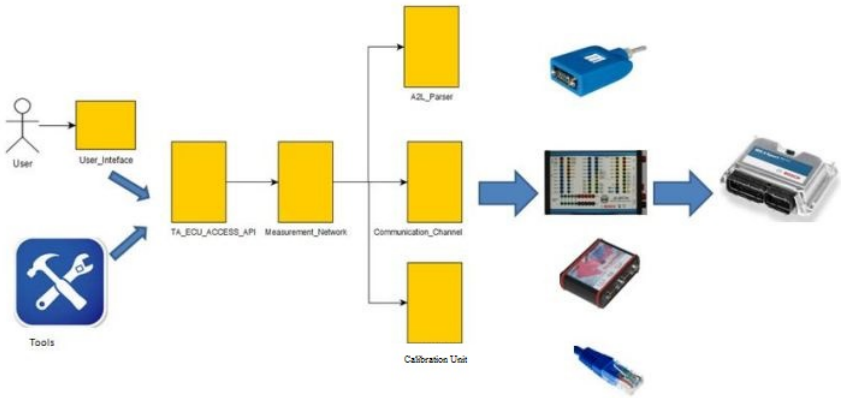


FIGURE 1.2: Overall Block Diagram

API Manager:

The end result of the system is an API which can be easily integrated into the test automation system. To make things easier for the API users, an API Manager which can take care of multiple users and at the same time maintain their state has been made by answering the RQ4.

A2L Parser:

The parser unit is responsible to extract the requested protocol related information from the given A2L file. This information will be later used by the protocol manager unit to initialize its data and prepare for communication with the ECU and thereby answering the RQ1.

Protocol Manager:

This unit is mainly comprised of the modules related to the calibration protocol which establishes communication between the ECU and the calibration device. All the units have been kept as independent as possible so that it is easier to extend or replace them with different protocols thereby answering the RQ2.

Hardware Driver:

This unit has access to the hardware which can send the requests provided by the protocol and adapts them to hardware related commands and communicates with the hardware. It is also responsible for receiving the incoming response and redirect it to the calibration unit. This unit is also easily replaceable or extended there by answering the RQ3.

Calibration Unit:

This unit receives the raw data from the protocol manager and responsible for reconstructing the signals with data. It can handle a huge load of incoming response and process them in parallel.

The following chapters will discuss about these blocks and how they are implemented.

1.4 Organization of thesis

The main focus of this thesis is to design the architecture of the tool using the requirements and research questions. This book has been followed [10] for solving most of the research questions. It has helped to create a design which is flexible, maintainable and can cope with the changes.

The thesis report has been organized as follows:

- Research Approach and Questions
- Detailed explanation about CCP and XCP
- Current existing solutions for calibrating ECU's parameters
- Detailed explanation of the architecture of design.
- Implementation of the identified solution.
- Validation.
- Future work.

Chapter 2

Research Approach and Setup

This chapter will discuss about various research apporach that have been taken to solve the problem discussed in previous chapter. Several meetings have been conducted with the technical experts to identify the selection process for the research. The main aim was to select a calibration protocol and a calibration device to measure the signals in ECU. To understand the result of meetings conducted, it is necessary to have an idea about the calibration protocols and devices available.

2.1 Calibration Protocols

The following are some of the protocols that are defined by the consortium of automotive industries like Continental Automotive GmbH, dSPACE GmbH, ETAS GmbH, M&K GmbH, Robert Bosch GmbH, Softing AG and Vector Informatik GmbH[2]. These protocols are defined in a file type called "A2L" files with *.a2l as extension. Based on the type of ECU configurations corresponding A2L files are generated. In the following chapters, A2l files will be discussed in detail. The detailed explanation of calibration protocols will be discussed in the following chapters.
Calibration protocols are nothing but a set of defined rules which has to be followed by the tester in order to calibrate the internal parameters of the ECU. CCP,XCP and ETK are the three calibration protocols which has been discussed here.

CCP:

This protocol works on the CAN bus protocol which uses CAN physical channel for data measurement acquisition, calibration and flash programming activities.

XCP:

XCP protocol is a successor of CCP. The main advantage of this protocol is that it can work on different transport layers like CAN, LIN, Flexray, Ethernet and many others. XCP provides higher performance and optimal resource utilization than CCP but XCP is fairly new and many ECUs do not support this protocol.

ETK:

This protocol is based on Ethernet. Maintained by ETAS GmbH and supported by devices from ETAS GmbH[2].

2.2 Calibration Devices

These are the devices which can communicate to the ECU for writing or reading values to and from the controller. There are many manufacturers for these devices in the market. For e.g. Vector Informatik Gmbh, ETAS Gmbh, Samtec Gmbh etc.
The below listed are some of the calibration devices that are predominantly available in the Bosch Laboratories and can be reused.

CANcaseXL:

This device shown in Figure 2.1 is manufactured and supplied by Vector Informatik GmbH. This device provides access to CAN physical channel which can create CAN Frames on request to transmit and receive data from CAN Bus. It comes with a well documented API which allows the user to set the Baud rate, filters and other channel related parameters through the API library[3].

FIGURE 2.1: CANcaseXL Device

Source: http://home.mit.bme.hu/ toth/boschweb/tools/CANcaseXL.jpg

ES581.4:

FIGURE 2.2: ES581.4 Device

Source: http://www.etas.com/data/products_Compact_ES500_ECU_Bus_Interfaces/PP_ES581_4.jpg

This device shown in Figure 2.2 is manufactured by Intrepid control systems and supplied by ETAS GmbH[8]. This device also provides access to CAN physical channel which can create CAN Frames on request to transmit and receive them from CAN Bus. It also comes with a well documented API which allows us to set the Baud rate, filters

and other channel related parameters through the API driver library provided by the Intrepid control systems. This device is cheaper compared to CANcaseXL.

ES600:

This hub shown in Figure 2.3 is manufactured and supplied by ETAS GmbH. It supports multiple channels and gives access to CAN physical channel and ETK physical channel. The API provided by the manufacturer fails to meet the expectation in terms of calibration performance. It is very expensive compared to the other devices mentioned here but there is a high availability of these devices in the Bosch laboratories[9].

FIGURE 2.3: ES600 Hub

Source: http://www.etas.com/data/group_subsidiaries_germany/ES593_1-D_rdax_630x422_100.jpg

μLC Test System:

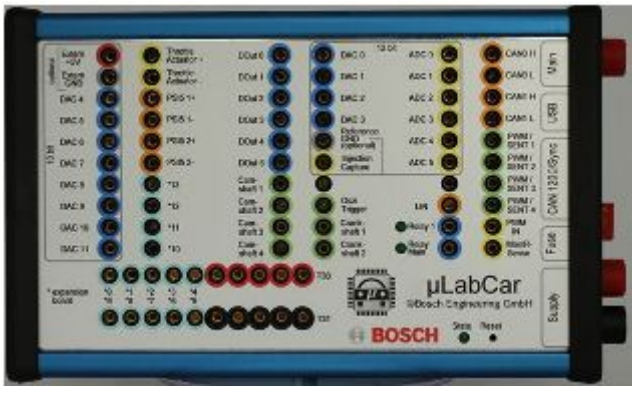


FIGURE 2.4: μLC Test System

This device shown in Figure 2.4 is manufactured and supplied by Bosch Engineering GmbH. This device supports 2 CAN channels and gives access to CAN physical channel. The API is currently under development. The availability of this device is high and comparatively cheap as it is built in Bosch Engineering GmbH.

2.3 Communication Overview

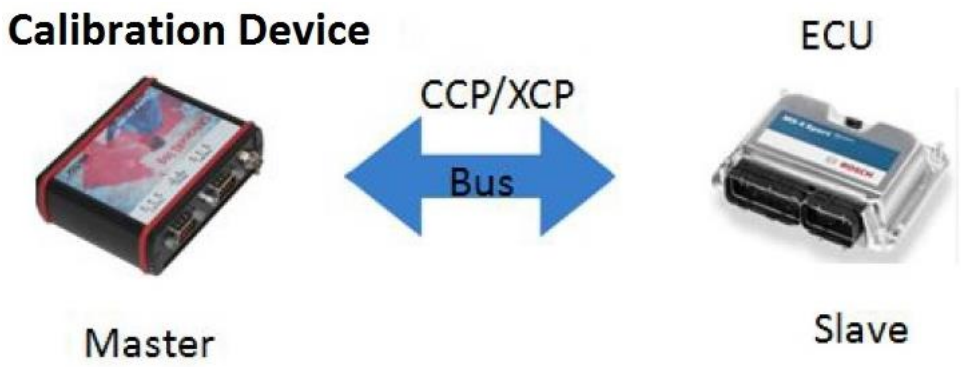


FIGURE 2.5: Master-Slave Communication

The Figure 2.5 shows the overview of communication with the ECU through the bus system. Here the calibration device acts as master and sends the command codes defined in the calibration protocol to start the communication with the ECU. If the ECU supports the requested calibration protocol then it acknowledges the same and starts the communication.The tester tool will act as the master and send the commands to ECU. The ECU acts as the slave which processes the requests sent by the master and gives the requested information back.
Here bus could be CAN, Flexray, LIN, Ethernet etc. But in this thesis only CAN bus has been used.

2.4 Weighted Sum Approach

Now an overall idea about the available protocols and calibration devices have been discussed. It is important to select one for starting the implementation.Several discussions with technical experts and users have been made to come up with selection criteria. The following weights to proceed with weighted sum approach have been identified.

Weight	**Assignment**
3	*Good*
2	$Moderateg$
1	*Bad*
NA	$NotAvailable$

2.4.1 Protocol Selection

To chose the correct calibration protocol following two important criteria has been taken.

Documentation: First criteria is that there should be proper documentation of the protocol to understand the protocol. Also , this will help in understanding the communication mechanism i.e. the list of command codes to be sent from the master device to which the slave reacts and gives the corresponding response.

Support: Second criteria is the available support of on the calibration protocols. ECUs come from different OEM and have their own architecture and memory layout. All the ECUs may not support all the calibration protocols.

Table 2.1: Calibration Protocol Selection

Protocol	Documentation	Availability	Sum
ETK	1	3	4
CCP	3	2	5
XCP	3	2	5

The table 2.1 shows the results of discussion made on the protocol selection. To give an overview of the results, key points will be discussed here. **ETK** is supported in many ECUs widely but no documentation is furnished by the ETAS GmbH for using that protocol. **CCP** and **XCP** has a very good documentation available and most of the ECUs support this protocol.

2.4.2 Device Selection

For device selection following criteria should be considered.

Cost: How much the calibration device cost and is it worth the cost to buy them?

Availability: Can the currently available devices be reused?

API Functionality: Good API will be one of the very important deciding factors, as API needs to communicate with the device and manage the device for custom suited needs.

Documentation: It is a basic requirement to have a good documentation, so that the users can easily understand the functionalities provided by the device and use them.

Support: As these devices are provided by the external manufactures, it is expected to have a good and quick response from the supplier to solve these concerns.

Table 2.2: Calibration Device Selection

Device	Cost	Documentation	API	Support	Availability	Sum
CANcaseXL	2	3	3	3	2	13
μLC	3	2	NA	3	3	11
ES600	1	2	1	1	3	8
ES581.3	3	3	3	1	2	12

The table 2.2 shows the selection favoring towards the CANcaseXL. The main problem with ES600 is that it is expensive and the API exposed by it could not handle the primary requirements. Nevertheless the design must be totally independent of the selected hardware and must be easily replaceable in future with other devices or to be connected in parallel.

Chapter 3

Protocol

This chapter will explain about the calibration protocols that have been used in this thesis.

3.1 CAN Calibration Protocol

3.1.1 CCP Basics

CAN Calibration Protocol widely known as CCP consists of a set of commands which should be sent from development tool to the ECU in order to measure, calibrate the internal parameters or for flashing the software. But there is a limitation on choosing physical layer while using this protocol. User can only communicate via CAN as no other physical layer is supported in this protocol. It is not compulsory to implement the entire protocol. Makers have very wisely categorized all the commands into mandatory and optional.So, depending on the requirements developer will implement the commands.[4] Ingineurbüro Helmut Kleinknecht was the creator of CCP. After it gained popularity, ASAP has taken over the responsibility to develop this protocol. Later they provided with many optional functionalities to the protocol. Although CCP contains a word "Calibration" but this doesn't signifies that the protocol is only used for the calibration of the internal parameters. It is widely used for measuring the signals, flashing the ECU with software etc. The tester can read following with the help of CCP:

- RAM
- PORTS
- ROM

- FLASH

The tester can write to following with the help of CCP:

- RAM
- PORTS
- FLASH

CCP is a Master Slave communication protocol. As shown in the figure 3.1, master and slaves are connected on a CAN bus. Master refers to the tester tool here. It means master will be responsible for communication initialization with the slave. Master will send the command to the slave and in return the slave will respond accordingly. This command is called as "Command Receive Object". Slaves denote different ECUs. It is possible to operate on the internal parameters of different ECUs simultaneously.

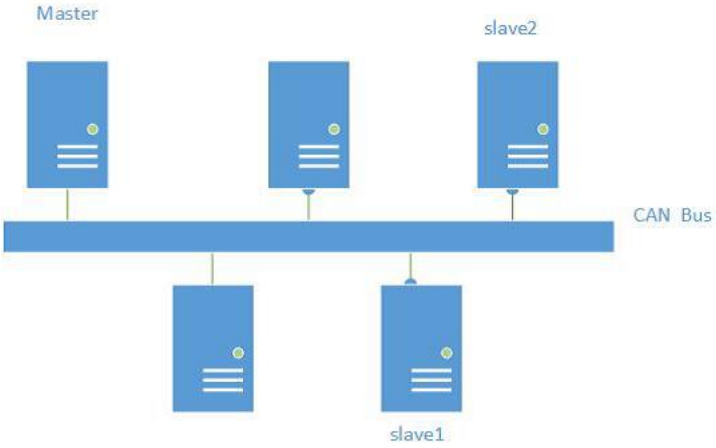


FIGURE 3.1: Master-Slave Device Configuration

3.1.1.1 CRO Message

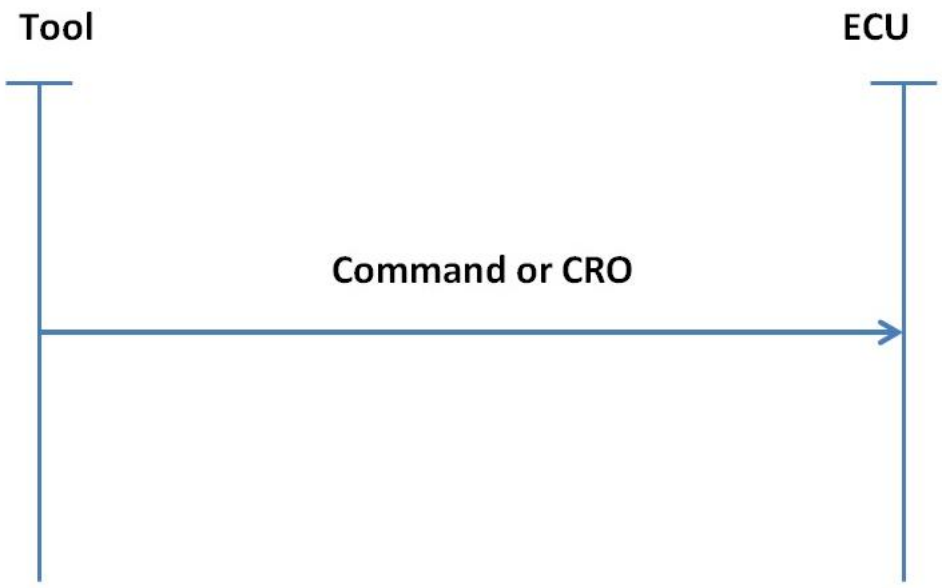


FIGURE 3.2: Command Receive Object

Source:[4]

As shown in the figure 3.2, CRO is the frame which is being sent to the slave by the Master. As CCP can only use the CAN physical layer, so data length of the message cannot exceed more than 8 bytes. As shown in the figure 3.3 CRO message has also 8 bytes, each byte having unique field.

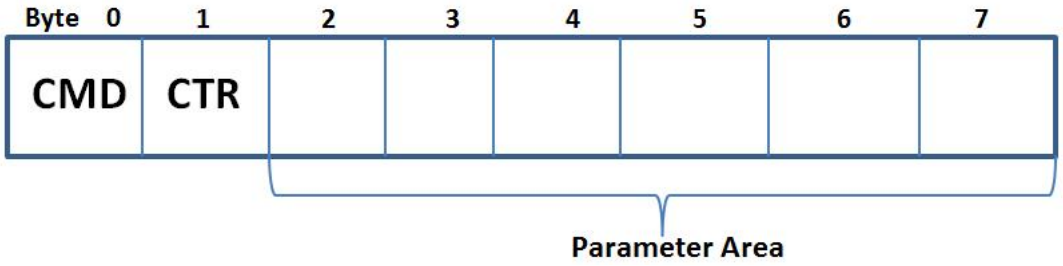


FIGURE 3.3: Structure of the CRO message

Source:[4]

As shown in the figure 3.4, first byte of the CRO message is a command number. Each command of CCP has a unique command number which is mentioned in the specification. The second byte is a command number counter from the tool to track the current

command that was issued. This same value will also be used in the response from the ECU to the tool.[4].

3.1.1.2 Data Transmission Object

Data Transmission Object is the frame sent from ECU to the master. Basically, this frame is the response of CRO message.

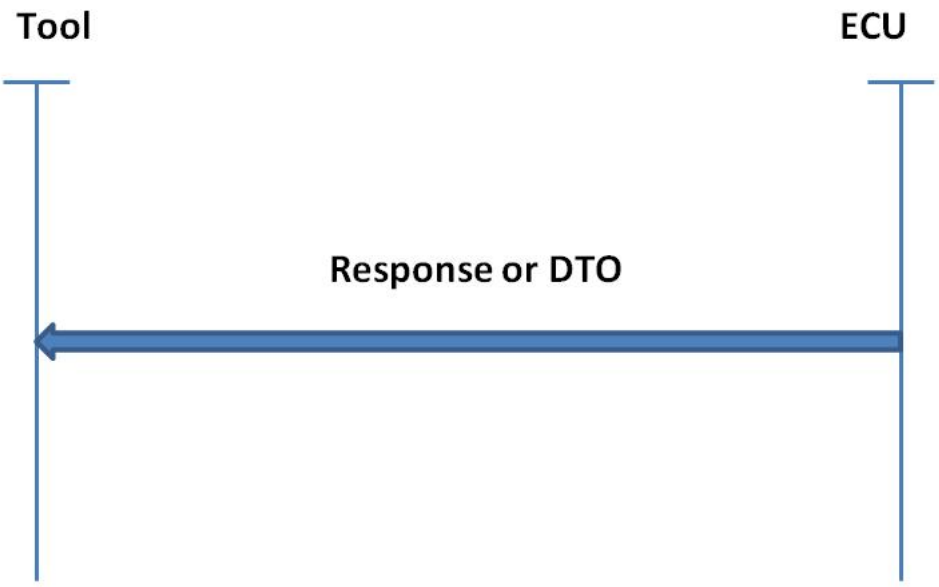


FIGURE 3.4: Data Transmission Object

Source:[4]

So, depending on the response DTO could of three types:

- Command Return Message (CRM)
- Event Message
- Data Acquisition Message (DAQ)

As, the name suggests CRM is ECU's response with respect to the CRO. For e.g when the master sends "Connect" command to the slaves. Then slave will send a CRM stating whether it is available for communication or not.

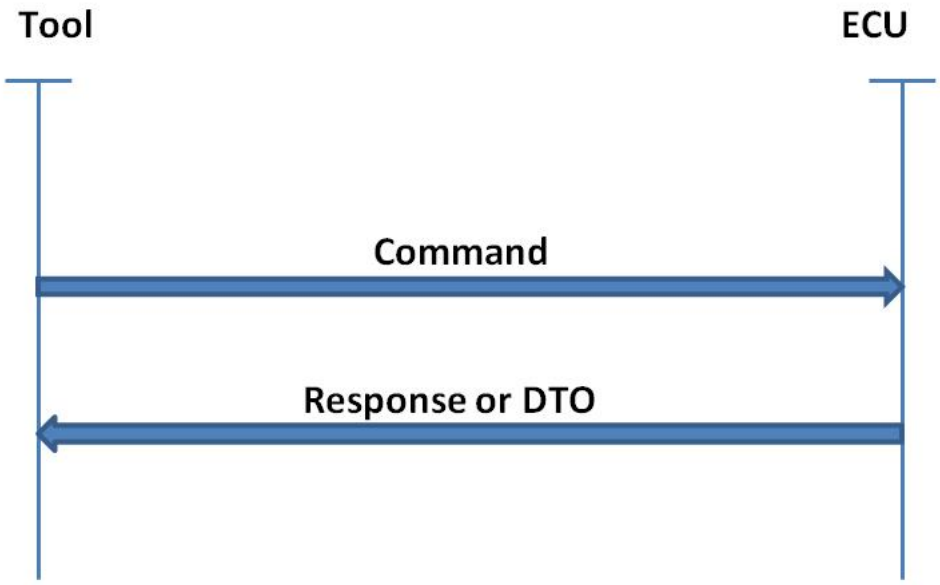


FIGURE 3.5: Command Return Message

Source:[4]

Event messages are used to denote errors in the tool. If the occurred events somehow changes the internal status of the ECU after master has sent the CRO then the master needs to be informed about that. This information is conveyed by the event messages.

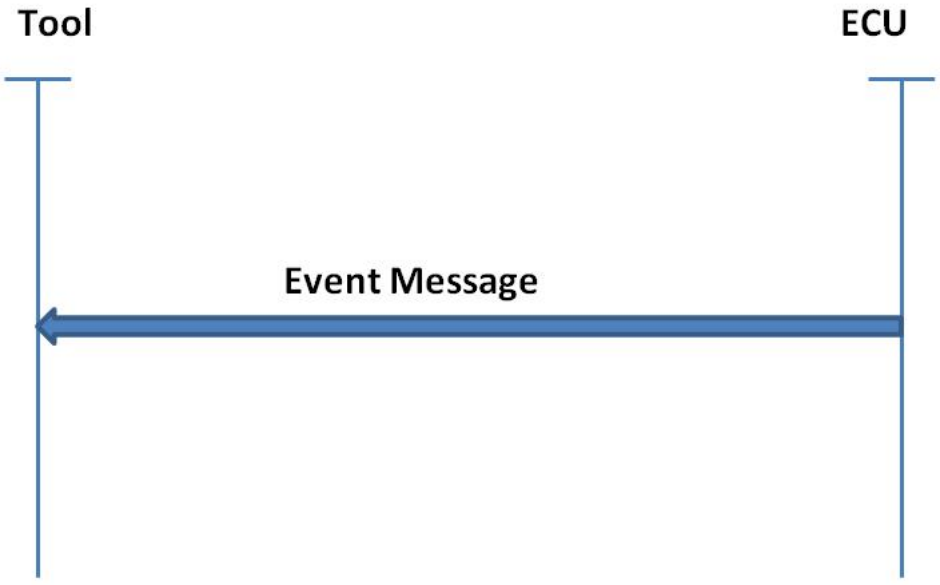


FIGURE 3.6: Event Message

Source:[4]

The first three bytes of both these messages have same structure. The first byte is PID i.e. Package Identifier. The PID for CRM is "0xFF" while that for Event message is "0xFE". Error code is denoted by the second byte. Command counter which has been sent in the second byte of the CRO will be returned in the third byte.

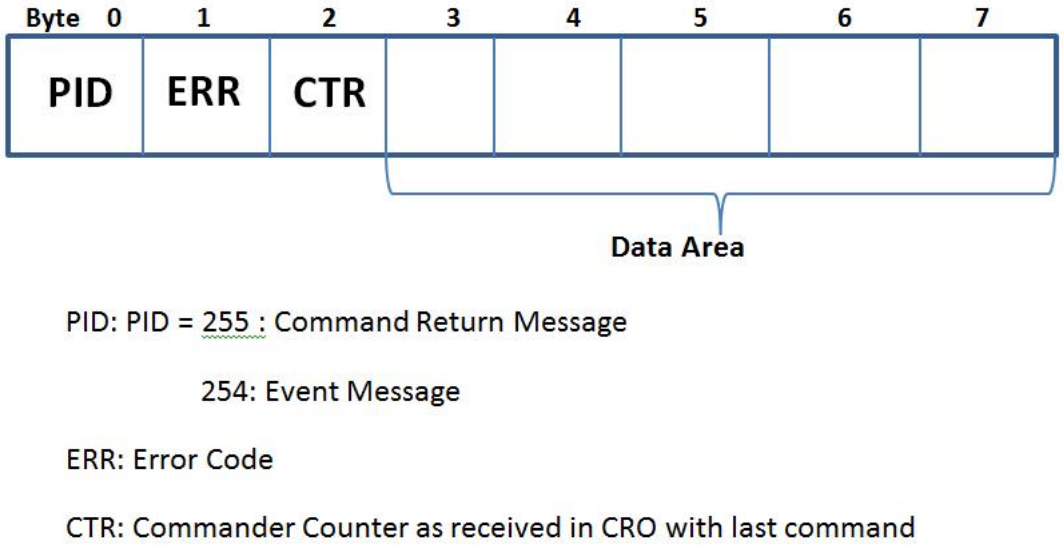


FIGURE 3.7: Structure of CRM and Event Message

Source:[4]

Data Acquisition message (DAQ) is a periodic message which will be sent to the ECU or Master at appropriate time. The only condition is that the ECU should be initialized. These messages are normally used for measuring the signals. As this thesis discusses about Calibration of internal parameters, so DAQ messages have not been discussed in detail here.

3.1.1.3 CCP Communication

The figure 3.8 gives an overview of communication between master and slave devices. The direction of different messages can be deciphered from the above figure. As explained earlier, the master will send a CRO to the slave. The slave will respond back with the DTO. All these message objects uses CAN as their physical layer [4].

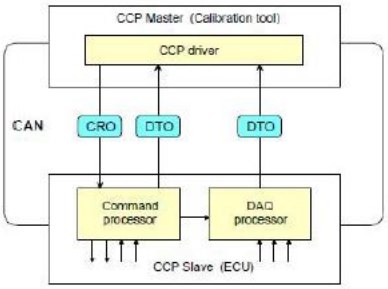


FIGURE 3.8: Message Object Communication

Source:[4]

3.1.2 CCP Commands

Although CCP offers many different commands for measurement, calibration, flashing etc. But in this section only the commands which is used for calibration via CCP have been discussed.

CONNECT:
This is the very first command which must be sent to the slave by the Master. If and only if the slave responded with a positive PID then only further communication will be proceeded.

Example: **Connect Frame: 0x01 0x01 0x00 0x00 0x00 0x00 0x00 0x00**

The first byte is fixed PID code for Connect command according to CCP 2.1 specifications. The 2nd byte is command counter which has to be returned in the response.

Positive Response: 0xFF 0x00 0x01

The 1st byte is the positive ID which is 0FF. In case of negative response this ID will be 0xFE. As it can be seen that the third byte of the response is same as the second byte of the CRO. This is very useful while tracking back CRO from the response.

EXCHANGE_ID
This command exchange IDs between master and slaves in order to set up the automatic session configuration.

Exchange_ID Frame: 0x17 0x02 0x00 0x00 0x00 0x00 0x00 0x00

Positive Response: 0xFF 0x00 0x02

GET_ACTIVE_CAL_PAGE

This command will returns the start address of the Calibration page that is currently active in the slave device.

GET_ACTIVE _CAL_PAGE Frame: 0x09 0x05 0x00 0x00 0x00 0x00 0x00 0x00

Here: 1st Byte: Command Code = GET_ACTIVE_CAL_PAGE 0x09

2nd Byte: Command Counter

3..8 Byte: Don't Care

Positive Response: 0xFF 0x00 0x05 0x00 0x12 0x34 0x56 0x78

UPLOAD

This command is used to get the specified size of data from the ECU memory. So basically this command will upload data to the master from the ECU. After writing data the current MTA pointer will now refer to the sum of previous address and the specified size. A maximum of 5 bytes can be uploaded at a time.

UPLOAD Frame: 0x04 0x06 0x04 0x00 0x00 0x00 0x00 0x00

Here: 1st Byte: Command Code = UPLOAD 0x04

2nd Byte: Command Counter

3rd Byte: Size of the data block

4..8 Byte: Don't Care

Positive Response: 0xFF 0x00 0x06 0x00 0x11 0x7C 0x82

It can be seen from the response that as requested in the frame, 4 bytes of data i.e. 0x00117C82 has been sent to the master from the ECU.

DOWNLOAD

This command is used to download the specified size of data to the ECU memory. So basically this command will write data from the master to the ECU. After writing data the current MTA pointer will now refer to the sum of previous address and the specified size. A maximum of 5 bytes can be written at a time.

DOWNLOAD Frame: 0x03 0x07 0x04 0x00 0x11 0x7C 0x82

Here: 1st Byte: Command Code = DOWNLOAD 0x03

2nd Byte: Command Counter

3rd Byte: Size of the data block

4..8 Byte: Data

Positive Response: 0xFF 0x00 0x07

3.1.3 Calibration using CCP

This section will explain briefly the process of doing calibration using CCP. There are two types of memory in the ECU:

- RAM
- FLASH

Depending on the definitions in the source code any parameter will be stored either in FLASH or in RAM. If the parameter is defined as constant then while linking and compilation of source code these parameters will be stored in the FLASH memory. The value of parameter can only be changed in the source code. Once the changes are made, the code has to recompiled again and then flash again to the ECU. This is a very tedious process. Another way of doing the calibration is changing the values in hex file directly and then flash it again. This is called as "Offline Calibration" [4]. Another method of calibration is the "Online Calibration". In this method the value of the parameters can be updated during run time. All the parameters whose values can be updated during run time are stored in RAM with the initial values stored in the Flash memory. The initialization is done while the ECU is getting booted. So all the parameters which are stored in the RAM are initialized to their initial values stored in the flash memory, as shown in the figure 3.9.

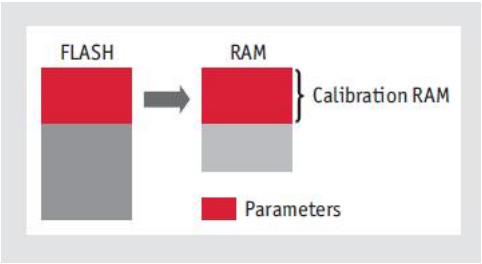


FIGURE 3.9: Initial Parameter setting in RAM

Source:[4]

This thesis is mainly focused on "Online Calibration". Suppose a calibration variable named has to be calibrated. Then following set of commands need to be sent from the master in the same sequence as shown in the figure 3.10:

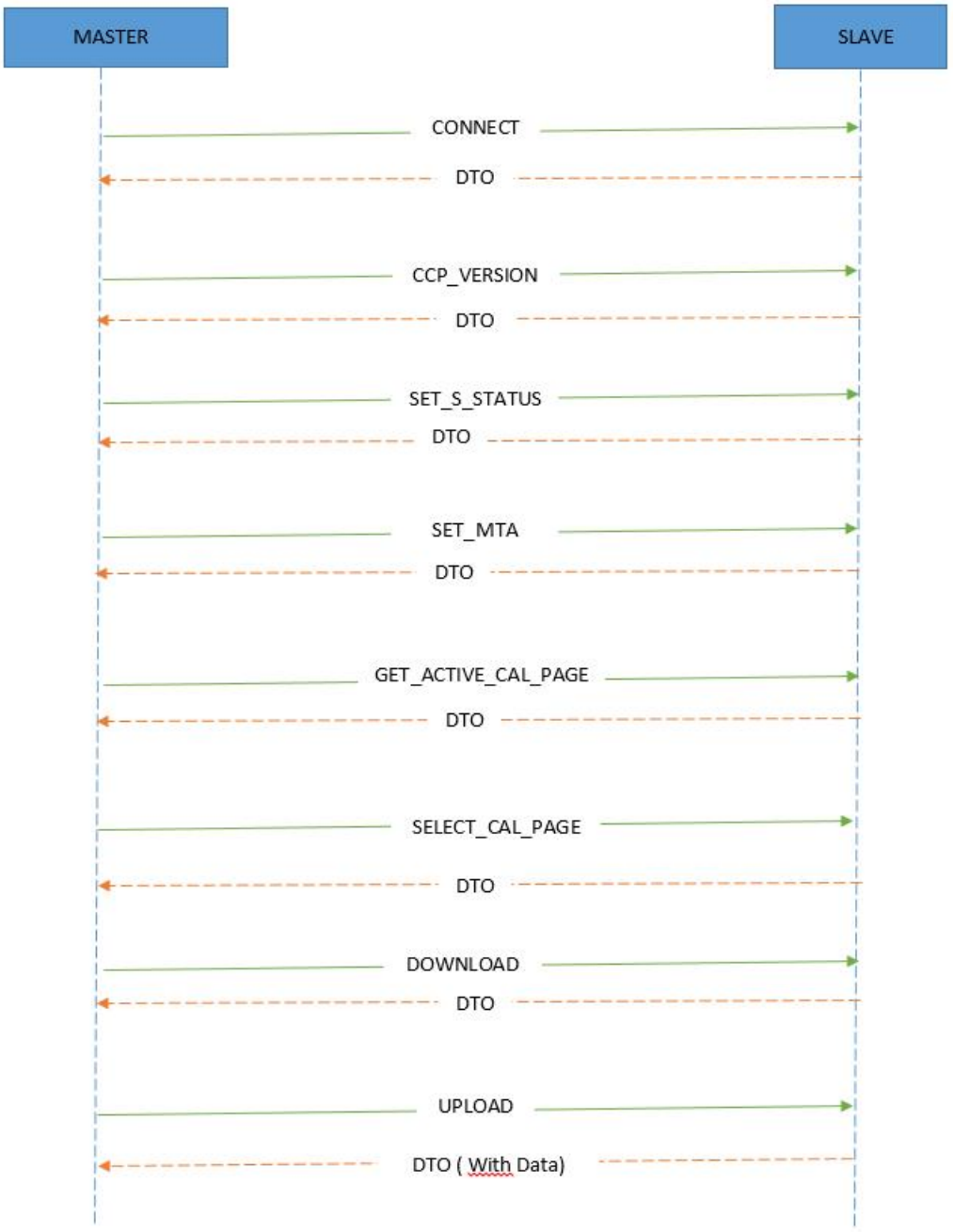


FIGURE 3.10: Set of commands sent for Calibration using CCP

3.2 Universal Calibration Protocol (XCP)

3.2.1 XCP Basics

"X" here stands for variable and interchangeable transport layer. It is similar to CCP but in contrast to CCP it can work on CAN, Flexray, Ethernet etc. So, the XCP protocol is independent of the transport layer. It is also a master-slave protocol [5].

FIGURE 3.11: Subdivision of the XCP protocol into protocol layer and transport layer

Source:[5]

This arises a question that how this protocol has been made independent from different transport layer. As shown in the figure 3.11, the XCP protocol is subdivided into protocol layer and transport layer. So, depending on the protocol layer information the protocol will choose the transport layer. This will make the protocol independent from a specific physical transport layer [5].

The tester can read following with the help of XCP:

- RAM
- PORTS
- ROM
- FLASH

The tester can write to following with the help of XCP:

- RAM
- PORTS
- FLASH

3.2.1.1 XCP Protocol Layer

The XCP Protocol layer is responsible for creating a XCP message frame. The message frame will carry the information about the transport layer (CAN, Flexray, Ethernet..). As shown in the figure 3.12, the frame is subdivided into three parts:

- XCP Header
- XCP Packet
- XCP Tail

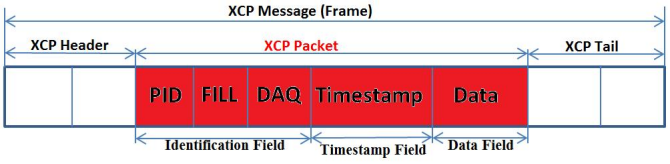


FIGURE 3.12: XCP Frame

Source:[5]

The XCP Header and XCP Tail contains the information about the transport layer, hence making the XCP packet independent from transport layer. As shown in the figure 3.12, there are three components in XCP packet which are:

- Identification Field
- Timestamp Field
- Data Field

"PID" in XCP packet stands for "Packet Identifier" which is used for packet identification. While exchanging messages the master and slave should able to identify the messages. This is done via PID. In XCP the PIDs have been defined as shown in the figure 3.13 From the below figure it can be seen that all the commands from master to slave has been accommodated in the Ids ranging from 0xC0 to 0xFF.

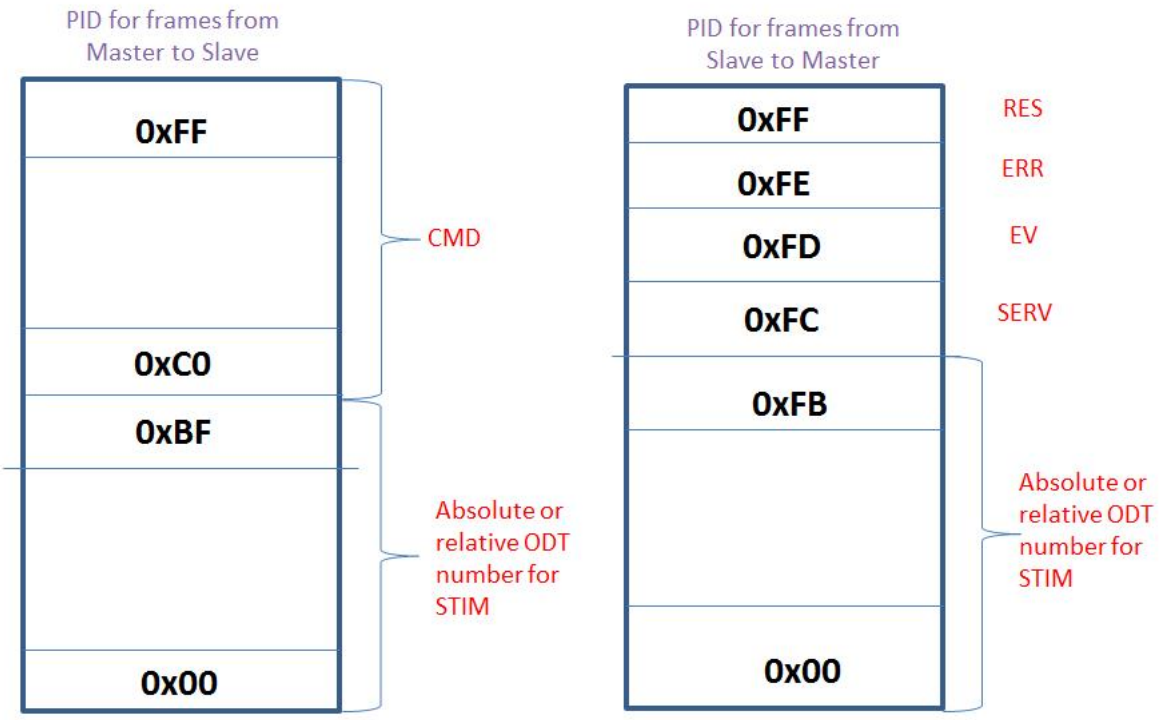


FIGURE 3.13: Overview of XCP Packet Identifier

Source:[5]

The expansion of acronyms and their purpose are shown in table 3.1:

TABLE 3.1: List of Acronyms used in XCP Communication

Acronym	Expansion	Description
CMD	Command Packet	Sends Commands
RES	Command Response Packet	Positive Response
ERR	Error	Negative Response
EV	Event Packet	Asynchronous Event
SERV	Service Request Packet	Service Request
DAQ	Data Acquisition	Send periodic measured values
STIM	Stimulation	Periodic stimulation of the Slave

Source:[5]

During calibration of internal parameters it is very important to know the time at which the calibration has been done. This information will always come from the slave (ECU) in the "Timestamp" field of the XCP packet. But transmitting time information by the

slave is optional. Data from the master or slave will be carried in the "Data" field of XCP packet [5].

3.2.1.2 XCP Transport Layer

The major drawback of CCP was that it can only uses CAN channel for communication. So when the notion of new protocol was conceived then the most important requirement was to make it available for different transport layers. Keeping that in mind the designers of protocol has defined it for following transport layers:

- XCP on CAN
- XCP on LIN
- XCP on Flexray
- XCP on Ethernet
- XCP on USB

As this thesis have used only CAN physical channel for communication using XCP, so only XCP on CAN is discussed in the report.Depending on the XCP header and XCP tail field the corresponding transport layer will be chosen.

XCP on CAN: As it has been known that the maximum data length of a CAN message is 8 bytes. But in XCP packet one byte is reserved for PID. So, only 7 bytes of useful data can be sent in one XCP message if XCP header and XCP tail fields are empty [7].

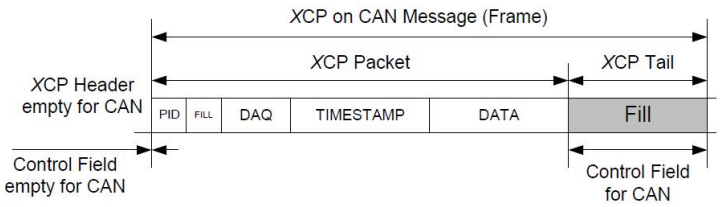


FIGURE 3.14: XCP Message for CAN

Source:[7]

As shown in the figure 3.14, XCP header field for XCP message on CAN is empty. The XCP tail field can contain Control Field (optional). Depending on the size of the XCP

tail, actual message size will be determined. For e.g. if XCP tail contains 2 bytes then the XCP frame will be structured like this:

- PID = 1 Byte
- Useful Data = 5 Bytes
- Tail = 2 Bytes

Generally XCP tail field will also be left empty for accommodating maximum data bytes. In the thesis while making XCP packets, XCP header and XCP tail has been left empty [7].

3.2.1.3 XCP Communication

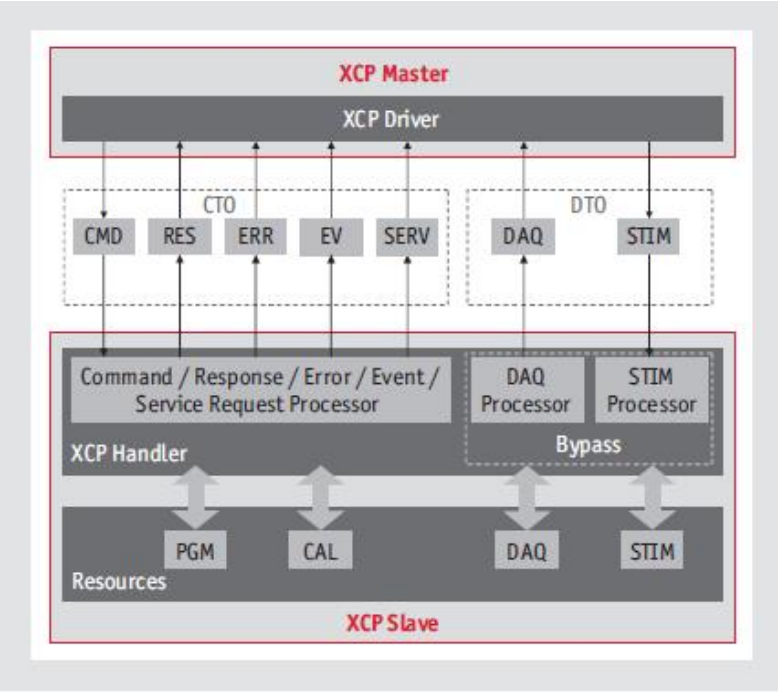


FIGURE 3.15: XCP Communication Model

Source:[5]

The figure 3.15 gives an overview of communication between master and slave devices. The direction of different messages can be deciphered from the above figure. The master will send a CMD to the slave. The slave will respond back either with the RES or with ERR.

3.2.2 XCP Commands

Again, although there are many XCP commands available, commands only useful for the calibration have been discussed.

CONNECT

The "Connect" command is used for setting up the connection with slave.The slave can either give a positive response or a negative response.
Please refer Appendix D for more details on the positive and negative responses of this command.

Table 3.2: XCP Connect Command

Position	Type	Description
0	BYTE	Command Code = 0xFF
1	BYTE	Mode 00 = Normal 01 = user defined

Source:[6]

GET_STATUS

The slave's current status information will be returned in the response of this command.
Please refer Appendix D for more details on the positive and negative responses of this command.

Table 3.3: XCP GET_STATUS Command

Position	Type	Description
0	BYTE	Command Code = 0xFD

Source:[6]

UPLOAD

This command will upload data from ECU to the master. The length of data bytes to be uploaded will be specified in the command.

Please refer Appendix D for more details on the positive and negative responses of this command.

Table 3.4: XCP UPLOAD Command

Position	Type	Description
0	BYTE	Command Code = 0xF5
1	BYTE	Number of Data Elements

Source:[6]

DOWNLOAD

This command will download data from the master to ECU. The length of data bytes to be downloaded will be specified in the command.

Please refer Appendix D for more details on the positive and negative responses of this command.

Table 3.5: XCP DOWNLOAD Command

Position	Type	Description
0	BYTE	Command Code = 0xF0
1	BYTE	Number of Data Elements
2	BYTE	1st Data Element
3	BYTE	2nd Data Element
7	BYTE	nth Data Element

Source:[6]

SET_MTA

This command will set the specified memory address in the ECU. Please refer Appendix D for more details on the positive and negative responses of this command.

Table 3.6: XCP SET_MTA Command

Position	Type	Description
0	BYTE	Command Code = 0xF6
1	WORD	Reserved
3	BYTE	Address Extension
4	DWORD	Address

Source:[6]

GET_CAL_PAGE

This command will return the current activated page in the ECU for calibration. There are different modes supported by this protocol:

- 0x01: ECU Access
- 0x02: XCP Access

Please refer Appendix D for more details on the positive and negative responses of this command.

Table 3.7: XCP GET_CAL_PAGE Command

Position	Type	Description
0	BYTE	Command Code = 0xEA
1	BYTE	Access Mode

Source:[6]

SET_CAL_PAGE

This command will set the current active page in the ECU for calibration if the page switching is supported in the ECU.

Please refer Appendix D for more details on the positive and negative responses of this command.

Table 3.8: XCP SET_CAL_PAGE Command

Position	Type	Description
0	BYTE	Command Code = 0xEB
1	BYTE	Mode
2	BYTE	Logical Data Segment Number
3	BYTE	Logical Data Page Number

Source:[6]

The 8 bits of mode parameters are described as follows:

- 0th bit: If Set the ECU can access the given page
- 1st bit: If Set the XCP driver can access the given page
- 2nd bit to 6th bit: Don't care
- 7th bit: If Set then the segment number can be ignored and the command is applicable for all the segments

3.2.3 Calibration Using XCP:

As discussed in section 3.1.3, there are two types of memory in the ECU which are RAM and FLASH. It has been also discussed that all calibrating parameters are stored in the RAM with their initial values being stored in the FLASH. So, in order to calibrate the parameters the master sends a set of commands to the slave (ECU). The slave responds back with the results. The detailed flow diagram of the sequence of commands and their responses are shown in the figure 3.16.

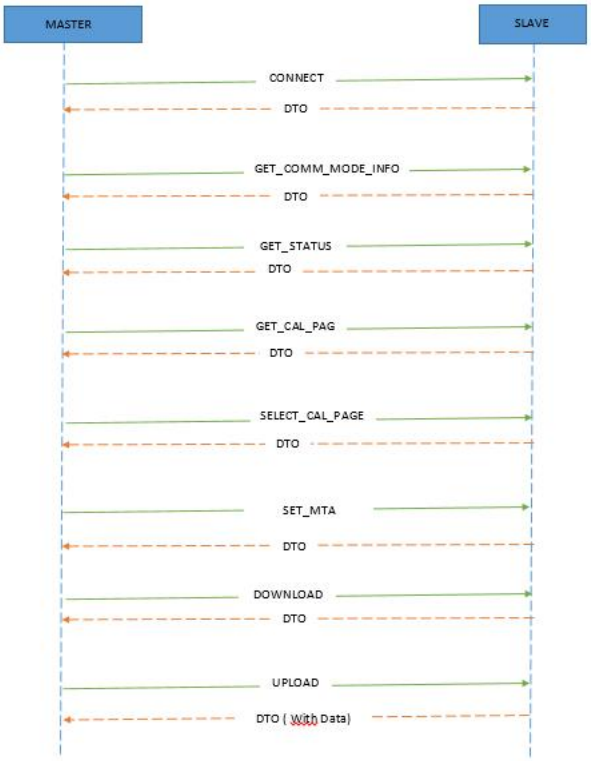


FIGURE 3.16: Set of commands sent for Calibration using XCP

As shown in the figure 3.16, the Master will send a "Connect" command first to the ECU. After successful establishment of connection the master will send "GET_COMM_MODE_INFO" which will return the different mode of communication supported by the slave. The next step is to find the current status of the slave. This can be achieved by sending "GET_STATUS" command to the slave. Then the current active calibration page is found out by sending command "GET_CAL_PAGE". If the page has not been set to the working page (RAM) then it has to be set via the SET_CAL_PAGE command. After setting to the working page, the memory address of parameter to be calibrated has to be set via SET_MTA command. Finally the value of parameter can be updated in the RAM via "DOWNLOAD" command. In case it has to be make sure that the value has been set in the RAM or not, "UPLOAD" command will be sent. The ECU should respond with updated value of the parameter.

Chapter 4

State of the Art

This chapter will discuss the current methods and tools which are used to calibrate the internal parameters of ECU using CCP and XCP on CAN bus. Discussion about the problems which user faces while using these methods and tools in an automated test setup will also be done in this chapter.

4.1 INCA

4.1.1 Introduction

INCA is a measurement and calibration tool which has been developed by ETAS Gmbh. Since many years this tool has been used in the industry for measuring and calibrating the internal parameters of the ECU. As shown in the figure 4.1, user can select the parameters of the ECU. These parameters are listed in a specific file called an A2l File. Please refer Appendix A to know more about calibration parameters present in the a2l file.

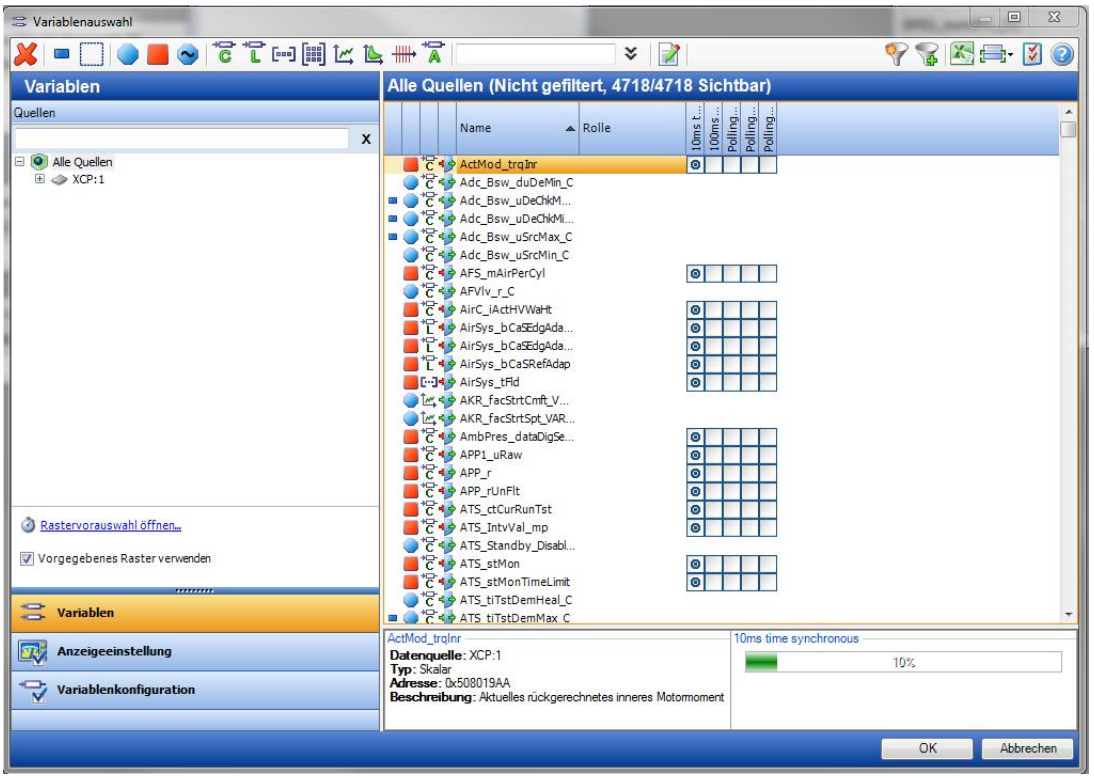


FIGURE 4.1: INCA Variable Selection Window

After the variables are selected, its values can be updated as shown in the figure 4.2.

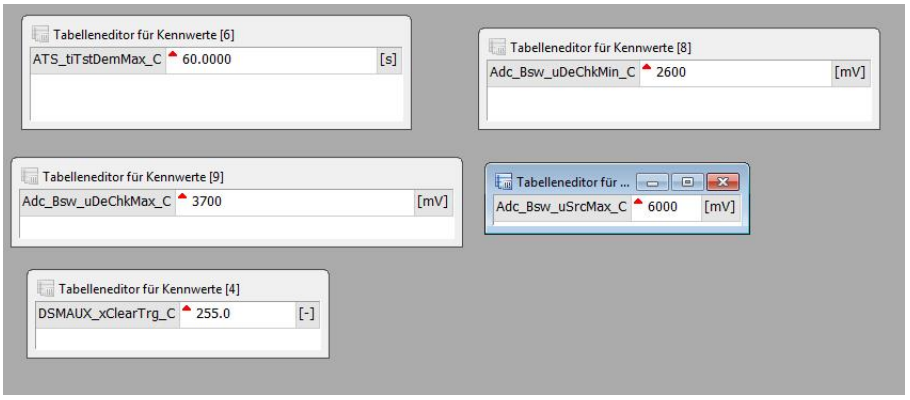


FIGURE 4.2: Calibrating Variables using INCA

INCA has several advantages:

- It supports parallel measuring and calibrating of internal parameters on multiple ECUs. This means that the values sent by ECUs to each other can also be

measured without any extra effort. This type of configuration of ECUs are called Master/Slave Configuration.

- It supports CCP, XCP, Ethernet protocols for measurement and calibration of internal parameters.

4.1.1.1 Problem with INCA

Although the tool is very robust, it has few problems.

- **Makes the Automation Process Slow:** When INCA is used for automating the test setup then it makes the whole set up very slow. To understand why it makes the process slow let's see the steps which INCA takes when calibrating the internal parameters.
  The very first step is to open the tool if the tool is not opened already. Opening INCA takes much time. The second step is to create experiment in INCA. Creating experiment includes adding a2l file, setting the hardware, adding hex file etc. This will take another some more time. The next step is to add the calibration or measurement variables into the experiment and then change the values. This will take some more time. During automation, time is very important.

- **Specific Hardware:** Hardware which will communicate to the ECU through INCA is only developed by ETAS Gmbh. This restricts the usage of INCA to only specific hardware in a limited amount of budget. If the hardware from different companies have to be used then user has to pay extra.

- **Costly:**The license for this tool is very costly. In addition to license, the user has to pay for the hardware also.

4.2 CCP Measurement Tool:

4.2.1 Introduction:

This is a tool developed internally by Bosch Engineering Gmbh. The notion of this tool was to overcome all the problems encountered in INCA. By using this tool, user can measure the internal variables using CCP. This tool is much faster than INCA.

4.2.2 Problems with the tool:

- **Master/Slave Configuration not Supported:** Measurement on multiple ECUs are not supported in this tool. Only the parameters of one ECU can be measured at a time. This thesis has solved this problem and will discussed in following chapters.
- **No Calibration Support:**This tool only supports measurement of internal parameters. So, user cannot change any internal parameter using this tool. This restricts the usage of the tool commercially.
- **Only CCP Supported:**As discussed CCP and XCP are two major protocols for measuring and calibrating internal parameters of the ECU. CCP can be used only for CAN bus while XCP can be used for several transport protocols such as CAN, Flexray, Ethernet etc. This tool supports measurement and tool only on CCP.

All these problems will be fixed with the solution provided in this thesis.

Chapter 5

Architecture

The main aim of this thesis was to design an architecture which is flexible, extendable and reusable without adding much effort. This chapter will discuss several design patterns which have been used while designing the architecture of tool.

5.1 Design Patterns

Design patterns are nothing but a well-defined solution to common design problems in software construction. Design patterns allow developers to share a common vocabulary for software interactions. [10]
Here all patterns which has been used in designing the architecture ahs been discussed. How that specific design has improved the architecture and answered the research question will also be discussed. There are three groups of design patterns:

- Creational Patterns
- Behavioral Patterns
- Structural Pattern

5.1.1 Creational Pattern:

There could be different situations when object of a class is getting created. This pattern takes care of such situation. So, basically this pattern group helps in object creation according to a specific situation.

5.1.1.1 Single point of Connection

Problem: Single point of Connection between API and the tool
APIConnector is a single point connection between User and the whole architecture. It has to be ensured that if any object of this class has been already created then user always gets that object. This is because if multiple objects of this class will be created then it is very difficult to track the requests from each object. This also make sure that when the tool is running then during the entire period of run time there is only one instance of Connector.
Solution: Singleton Pattern
Singleton Pattern is a very important pattern which have been used in the architecture design of the tool. "It ensures a class has only one instance and provides a global point of access [10]" This means once the tool is running there will be only one instance of singleton class. This can be achieved by defining constructor of the class as private and writing a public method to create the object of this class. There are two singleton classes in the architecture. As shown in the figure 5.1, there is a singleton class called "APIConnector" which can be accessed by the user.

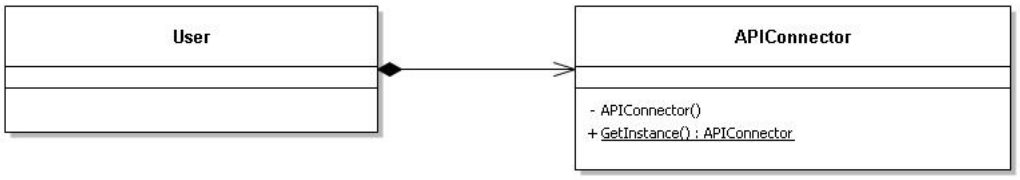


FIGURE 5.1: Singleton APIConnector Class

The second singleton class is *ECUAccessManager*. The connector will create an object of this *ECUAccessManager* to get information from the network. Purpose of this class is to manage all the information to the user via connector and hides the implementation details to the external user.

5.1.1.2 Processing Multiple ECUs simultaneously

According to the requirement, the tool should be able to process multiple ECUs simultaneously. But singleton class make sure that there is only one instance of the class. This is contradictory.

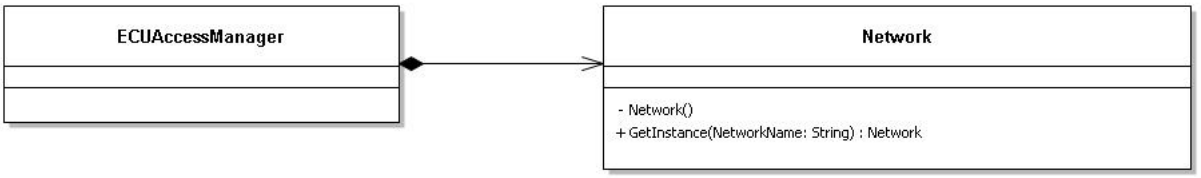


FIGURE 5.2: Multiton Pattern Example

Solution:Multiton Pattern

"The Multiton provides a method that controls the construction of a class: instead of maintaining a single copy of an object in an address space, the Multiton maintains a Dictionary that maps keys to unique objects"[23]. It should be noted that each calibration protocol should be referred as one network. As shown in the figure 5.2, user needs to pass the *NetworkName* as an argument to *GetInstance()* method. When user calls this method with name of the network then it checks the network name in a predefined dictionary. If the network name is already available in the dictionary then it will return the *Network* object corresponding to that network name. If the network name is not available then it will create a new *Network2* object and store it in the dictionary. So, there can have multiple singleton objects. This solves the problem of communication between multiple ECUs connected in parallel.

5.1.1.3 Extension to different Transport Layer

Problem: Extension of architecture to different Transport layer

This ia very important requirement (also mentioned in RQ2) to extend the architecture to different transport layer such as CAN, Ethernet, Flexray etc. Although this thesis only focusses on calibrating the internal ECU parameters only on CAN bus but it might be possible that in future the tool has to be extended on different transport layers. At that time the fames have to be adapted according to specified communication protocol. Aim is to make sure that this extension can be done with the minimal effort.

Solution: Abstract Factory

The above problem can be solved by applying factory pattern in the design.In Object oriented design, abstract classes are those class which cannot be instantiated with the "new" keyword. In order to implement these classes, it needs to be inherited in a new subclass. [23] Factory pattern are those which is used to create a family of related objects without explicitly giving the details of classes [23].

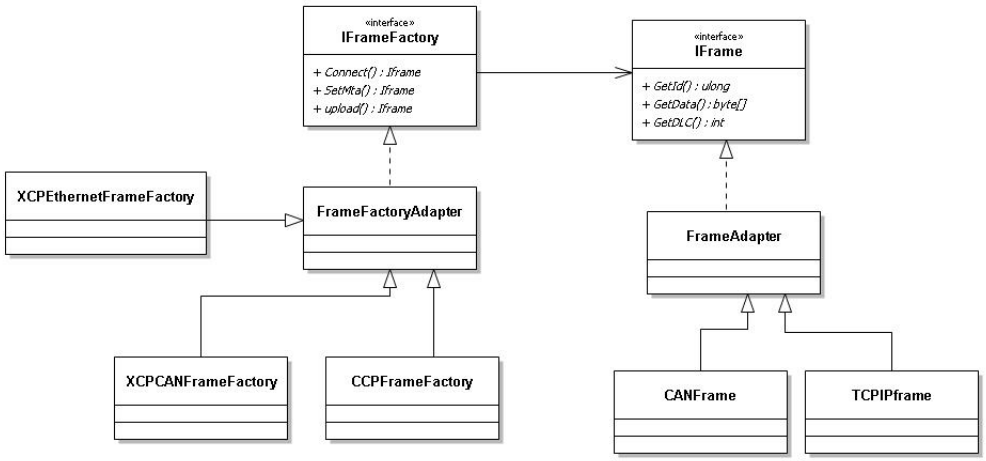


FIGURE 5.3: Abstract Factory Pattern Example

In the figure 5.3, *IFrameFactory* is a simple factory which contains all the CCP and XCP commands. It has to be kept in mind that the commands of CCP and XCP will be same irrespective of the transport protocol used. For e.g. if client wishes to connect to the ECU then he/she has to send the *CONNECT* frame irrespective of the bus used (CAN, Ethernet etc.). Only difference will be in the organization of frame. For Ethernet the TCP/IP frame will be created and for CAN bus CANFrames will be created. This can be decided in the run time depending on the protocol information given by the user. So, while creating a network, depending upon the protocol it will automatically calls *CCPFrameFactory*, *XCPCANFrameFactory* or *XCPEthernetFrameFactory*. *XCPCANFrameFactory* and *CCPFrameFactory* will implement all the methods defined in IFrameFactory and return *CANFrame* while *XCPEthernetFrameFactory* will return *TCP/IP* Frame.
With the help of this design RQ2 have been successfully answered.

5.1.2 Behavioral Pattern:

Behavioral patterns will take care of object responsibility and their interactions with each other.

5.1.2.1 Modular Architecture

Problem: Responsibility of sending commands should be moved to another class for better modularity of the architecture

It has been mentioned earlier in this chapter that the architecture should be easy to maintain. Maintainenance of a tool depends upon the structure of tool and much on modularity of the tool. So, it is needed that each class should hold single responsibility.

Solution: Template Method

This pattern deals with moving the steps of an algorithm or process to a sub class and accessing it through template method [24].

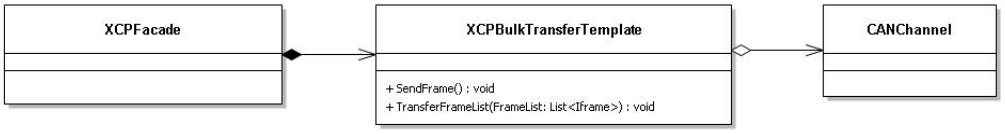


FIGURE 5.4: Template Method Pattern Example

The above figure 5.4 shows *XCPBulkTransferTemplate* class. The *XCPFacade* will call this class to send frames to the ECU. The facade will simply initialize this class and uses its reference to start the calibration. The method *TransferFrameList* will be called in the facade and all the frames which need to be send on the ECU will be passed as a list. It is the responsibility of *XCPBulkTransferTemplate* class to send the frame one by one and process the result. This helped in decreasing the complexity of the facade. The facade does not care about the implementation involved in the template method. This will provide more modularity in the architecture.

5.1.2.2 Simple Architecture

Problem: How to reduce the complexity of API in terms of looping over collection objects

Lesser the complexity, less effort to maintain the tool. As stated earlier, there are two deliverable products from this thesis. First is the tool with GUI and other is the API which can be directly integrated in the test environment. The architecture contains many dictionaries and lists which contain useful information. While integrating API into the test automation user has to know the type of dictionary or lists if it has to be used. Although this problem can be solved by using *foreach* loop and defining type as *var*, this is an old school method and less powerful. So there is a need of new pattern which user can use without knowing the type of collection object and meanwhile more powerful.

Solution: Iterator Pattern

This pattern allows the clients to access the elements of a collection object sequentially without the need to know its underlying implementation [10].

While delivering the API it has been made sure that all the collections are converted into the iterator. So, the client can simply iterate over the collection in order to get the object. Client doesn't have to worry about the underlying implementation of the collection.

For example, if there is an array which contains all the calibration variables selected by the user. User can then simply iterate over the collection using *MoveNext()* method. This method when called checks for the next element in the collection. If there is a next element in the collection then it will return true else it will return false. The user can then get the current object by calling *Current* property. This property will always return the current object set by the pointer. There is another method called *Reset* which will reset the pointer to the beginning of the collection. This is shown in the figure below.

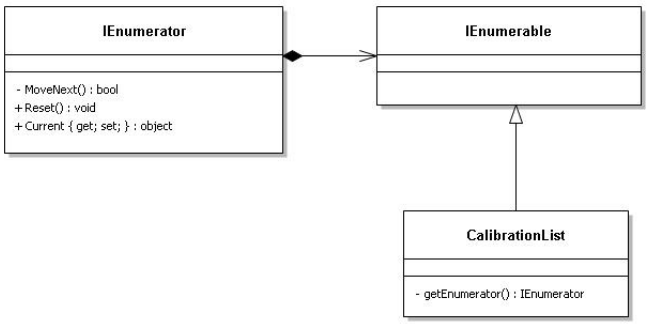


FIGURE 5.5: Iterator Pattern Example

5.1.2.3 Tracking Activities of User

Problem: To keep a track on the activities performed by the user

It might be possible that user tries to calibrate values without actually being connected to the slave. Also while debugging it should be easier if the design is following some sort of predefined state. This will make the process of debugging easier as the debugger can directly know at which step there is a problem.

Solution: State Machine

Implementing state machine can solve the above problem. State machine is defined as the "Change in the object behavior when its state changes" [24]. By this design it has been made sure that users are following the proper state flow and if not then a proper error message is showed to them. There are totally 4 states defined as shown in the figure below:

- **Initial State:** This state will be responsible for sending the Initialization frames such as *Connect*, *GetCommModeInfo* etc. Then the state is set to *CalibrationSelected* state.

- **CalibrationSelected State:** This state is responsible for setting all the selected calibration variables in a collection. Due to this state users will have the flexibility to add or remove few more calibration variables in the runtime. Then the state is set to *CommStarted* State.

- **CommStarted State:**This is the most important state because this state is responsible for sending all the calibration commands to the ECU. As shown in the figure below, there is a bidirectional arrow between *CalibrationSelected* and *CommStarted* State. This means if a new variable is added and the current state is *Commstarted* then it will jump to the *CalibrationSelected* state and add that variable to the collection. After modifying the collection the current state is again set to the *CommStarted* state.

- **CommStopped State:**At last in this state all the communication from the ECU will be stopped. If the user wants to get the value of calibration variables then an error message will be displayed to them.

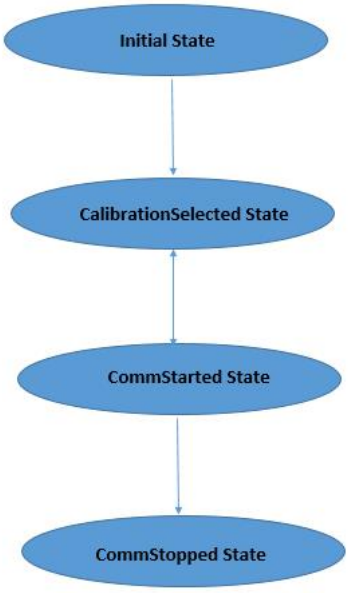


FIGURE 5.6: State Diagram

In the figure 5.7 it has been shown that how the state machine has been implemented in the design. If the user tries to get the value of any parameter in the initial state then an

exception will be raised stating that user have not selected the variable. The inclusion of state machine design has given more clarity in the architecture.

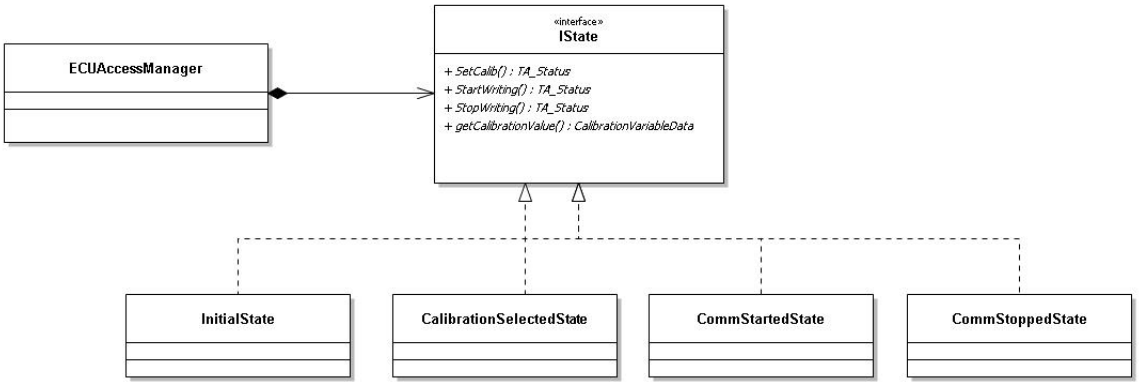


FIGURE 5.7: State Machine Class Diagram

5.1.3 Structural Pattern:

Structural Pattern explains about the composition of different classes to form a larger structure which can be further extended.

5.1.3.1 Exposure of logic to third party

Problem: How to access the system without exposing the implementation logic

As this tool is going to be used for commercial purpose, so it is very important to hide the implementation logic from the customer. Secondly, the architecture should be modular for better maintainability. Finally, the tool should able to hand multiple clients simultaneously.

Solution: Facade Pattern

Facade pattern is a very important pattern which can provide modularity to the architecture. Due to this pattern the *Network* can easily have access to the entire system without exposing the whole implementation logic. The facade gives a simplified access to a larger structure [10]. Each facade object holds the data parsed from a2l file, communication frames and the data received from hardware. So for each ECU there will be a new facade object. Each facade object will have all the information about that ECU. Each facade is mapped to a calibration profile. So if three ECUs are connected in parallel and all these ECUs wants to communicate with the tool then they can easily

do it without any ambiguity with the help of Facade and calibration profiles. Moreover the user does not needs to know about the complex internal logic behind the implementation.

As shown in the figure 5.8, the Network class communicates only with the *IFacade* interface. The *Ifacade* interface has the reference to the parser, FacadeAdapter, Channel etc.

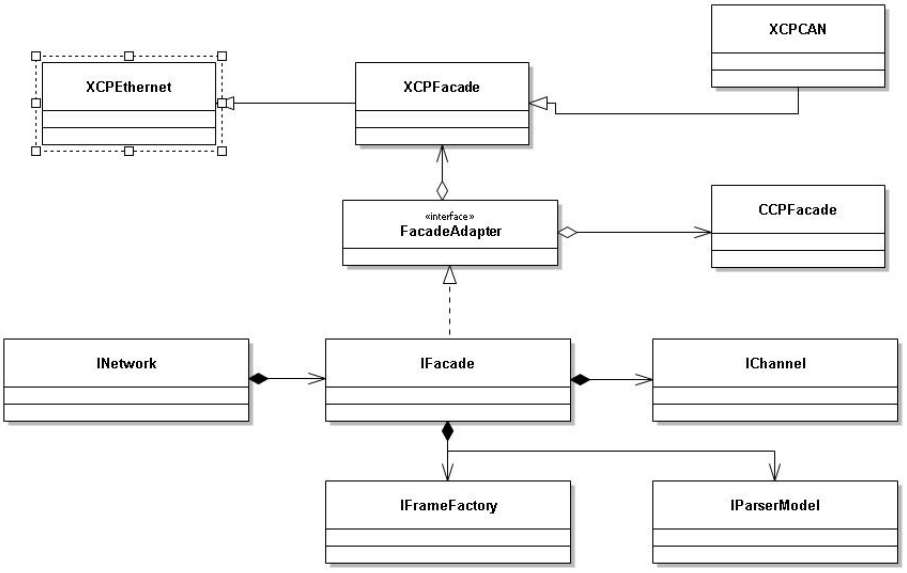


FIGURE 5.8: Facade Pattern Example

5.1.3.2 Extension to different Hardware

Problem: To extend the tool for multiple hardware from different vendors

As mentioned in RQ3 the tool should able to support hardware from different vendors. So, if in future a new hardware comes up and that has to be integrated in the solution then it should be done with minimal effort.

Solution: Adapter Pattern

Adapter Pattern is another beautiful pattern used by the architecture to make it work for different incompatible hardware interfaces.

The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces [24].

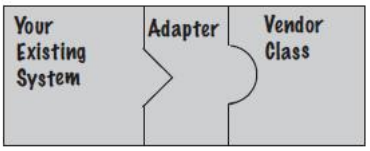


FIGURE 5.9: Adapter Pattern

As shown in the figure 5.9 while using the third party classes it might be possible that the interfaces provided by them is not compatible with the interface.Secondly, the tool should also support *MockUp* device. It means that the tool should not crash if there is no hardware connected while testing. This problem can be solved by creating a middleman or adapter which will receive the requests of the tool and convert it to the request compatible to the third party classes.
Now, the question is how it will be beneficial in the architecture. To understand that see the figure 5.10:

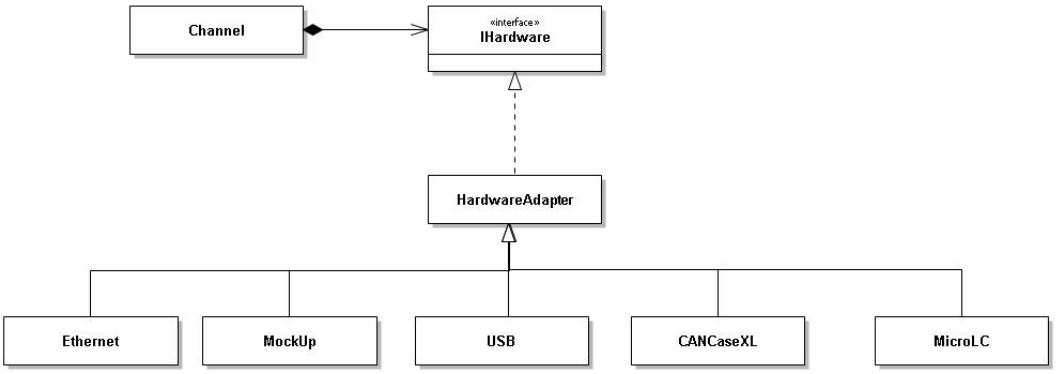


FIGURE 5.10: Dependency Inversion in Hardware Devices

It might be possible that the tool can be used with different hardware. The goal is to make a system which is independent of the hardware used. *CanCaseXl* is a hardware provided by Vector Informatik Gmbh. In order to use that in this system it should use the predefined classes provided by Vector Informatik Gmbh. Similarly, *MicroLC* is a hardware developed by Bosch Engineering Gmbh. This also have different APIs. In the above figure it can be seen that channel is only dependent on *IHardware*. With the help of *HardwareAdapter* the tool can easily convert the incompatible interfaces to the compatible interfaces which can be understood by both the hardware and channel. *HardwareAdapter* implements all the methods which is understood by the *Channel* class. MockUp Device is another feature which has been implemented in this design. An option

of virtual device has been given to the user called *MockUp* device. This can also be used for the demonstration purpose. When mockup device is detected by the facade then only log files will be written and system will not throw any exception.
This design has helped to answer RQ3

5.2 Architectural Style:

"An architectural style defines: a family of systems in terms of a pattern of structural organization; a vocabulary of components and connectors, with constraints on how they can be combined" [25]. An Architectural style is sometimes called as architectural pattern. The architecture of a complete system is never limited to a single pattern or style, but often a combination of architectural styles that makes up the complete system [25]. Usage of different architectural styles in the design benefits provided by it has been discussed in this section.

5.2.1 Easy to load User Interface

Problem: To make the UI as light as possible
As discussed earlier, opening INCA consumes much time as the GUI is very bulky and load so many things before finally getting opened. Keeping that in mind the architecture of UI has to be kept as simple as possible without disturbing the modularity of the design.

Solution: MVC Design Pattern

MVC stands for Model- View – Controller. The entire architecture is organized using this design.

- **Model:** Represents only data and it is independent from *Controller* or *View*
- **Controller** This is the core of the design. It acts as a communicator between *Model* and *View*. It takes input from the *View* and communicates with the *Model* to perform the operation and then update the *View* accordingly.
- **View** It is just the visual representation of the tool or system which will take inputs from the users and send this to the controller.

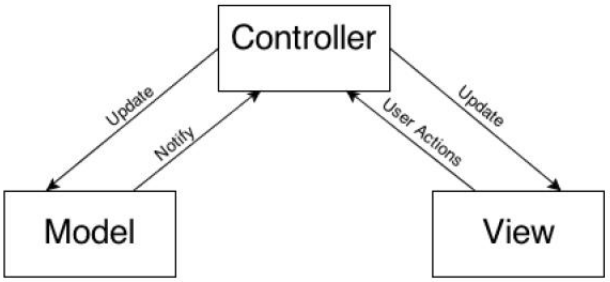


FIGURE 5.11: MVC Design Pattern

User actions will be done on the *View* i.e. User Interface. The actions will force the controller to update the *Model*. *Model* then notify the *Controller* that it has been updated. *Controller* then update in the user Interface.

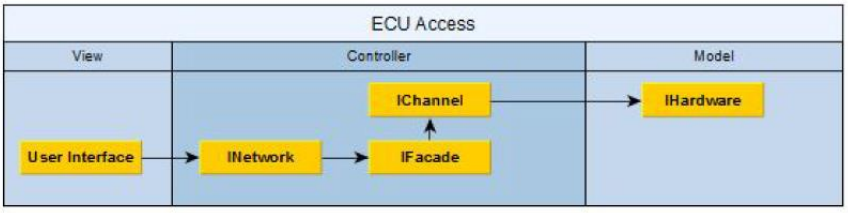


FIGURE 5.12: MVC Outline Diagram

In the architecture, User Interface will definitely come under the *View* part. Here it can be seen that *IHardware* class has been put in the *Model* part. This is because the hardware is totally independent from the Controller. User will send requests from the UI to the *Controller*. The *Controller* then process the requests and send it to the *IHardware*. The hardware responds back with the data form the ECU which the processed by the Controller to update the View.
The main advantage of this design is that *IHardware* class can be completely reusable. There is no dependency between *Model* and *View* hence the system can be easily extended without putting much effort. Secondly, new features can also added to the UI without worrying about the Hardware.

5.2.2 Handling Events

Problem: How to handle the events raised by multiple clients communicating in parallel.
Each ECU when connected in parallel raise multiple events after receiving messages.

All this messages should be organized in such a way that while decoding, the message should not mapped to another ECU and not to the ECU which has raised the event.

Solution: Pipe and Filter Style

The pipe and filter style is used in the system for continually and incrementally processing the data in the callback channel. As there are clients communicating in parallel, it is needed to pipe and filter the data at each level. The pipe-and filter style is applied to the runtime elements, so it is part of the runtime view type [28].

The Figure 5.13 shows how the raw data received from the hardware is processed and published in each class.

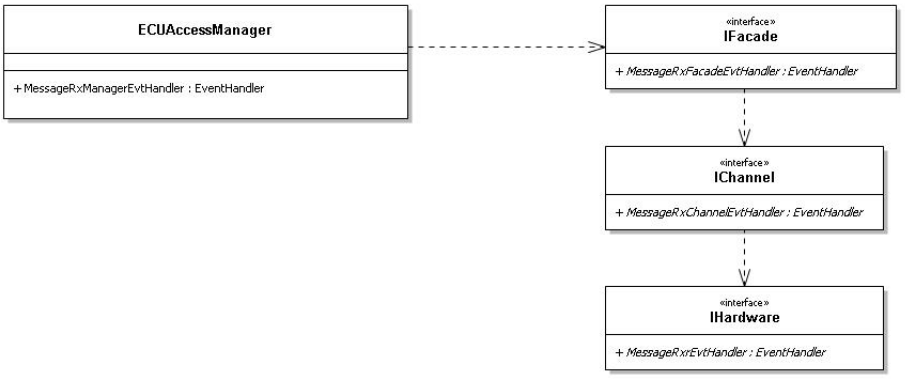


FIGURE 5.13: Callback Channel Class Diagram

- **IHardware:** This interface reconstructs the raw data into the selected protocol frames so that it can be processed by the upper layers.
- **IChannel:** This interface filters the data applicable to its channel based on the message identifiers. As it is possible that multiple virtual channels are connected to the same calibration device it is needed to filter and channel them to the corresponding facade classes.
- **IFacade:** This interface on receiving the protocol frames, extracts the data from each frame and reconstructs them into the proper data to map it into the calibration variable.
- **ECUAccessManager:** This class is responsible for handling multiple clients in parallel in a particular network. Based on the calibration data received from the facade it must identify the client and raise the event for that client.
  The clients after receiving the published data from the ECUAccessManager class

can filter and process them into ways which are convenient for plotting or calculating the measured data.
The event framework provided by the Microsoft .Net takes care of the write and read ports [28]. It is just needed to create instance of the object to listen to the events raised by them.

5.3 A2L File Parser

As per the RQ1 there should be a solution to parse all the information from A2l file to the data structures. Although there are many methods to parse a file a very powerful open source tool called *ANTLR* has been chosen to serve the purpose. *Regular Expressions* have also been used to extract the calibration protocol information from the A2l File.

Regular Expressions:
This is a most common method used nowadays to extract information from a file. The input for this tool will be a predefined expression. The tool will match the expression in the file to be parsed. If there is a match then that part will be extracted from the file.

ANTLR: "Regular Expression" technique to parse the entire a2l file cannot be used alsways. The reason being this will make the entire process slow. As the benchmark of the solution is the total time taken by the tool, so there is no chance to compromise in speed. Therefore, ANTLR have been used. ANTLR is an open-source tool targeted for programmers to develop data readers, language interpreters and translators. It is important to understand how the ANTLR works before going into the details[22].
ANTLR takes a grammar file as an input. Grammar file contains a set of rules which will define what all has to be extracted from the imput file. This grammar file when runs over the A2l file and generate few predefined classes which will be discussed in the next chapter.

5.4 Tool Block Diagram

The figure 5.14 shows the block diagram of the tool which has been achieved by using the above mentioned design patterns.

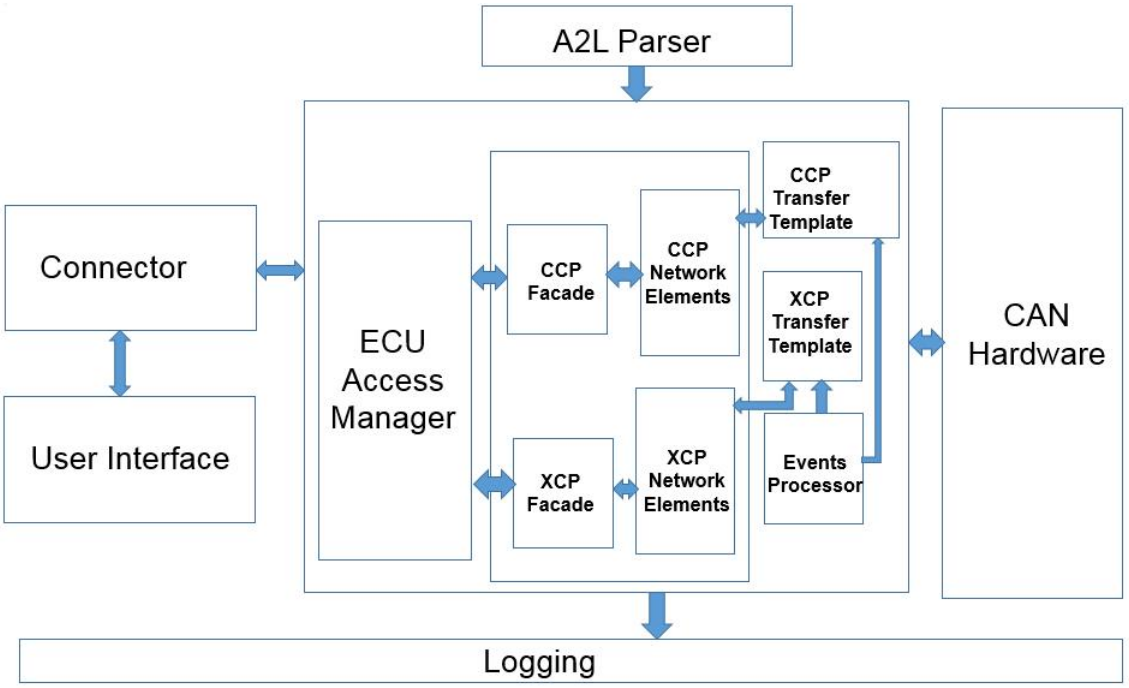


FIGURE 5.14: Block Diagram of the tool

All the problems which have been mentioned in the previous sections have been solved by using this block diagram. The detailed discussion on how it has solved the problem will be done in the next chapter.

5.5 API Strategy:

This section will discuss about the solution provided to communicate multiple clients in parallel. The discussion on how the API provided to the customer will mask the implementation logic of this system will also be done here.

5.5.1 Need for API Strategy:

Problem: Deliver the product without exposing the implementation logic to the customer
The main concern was to deliver a product which should be easily integrated in the test automation system without exposing the implementation logic to the cusotmer.
Solution: The problem can be solved by developing an API which will expose only selected information to the user. But it has to be to kept in mind that the API should not

hamper the ability of this product to communicate with multiple clients simultaneously as shown in the figure 5.15.

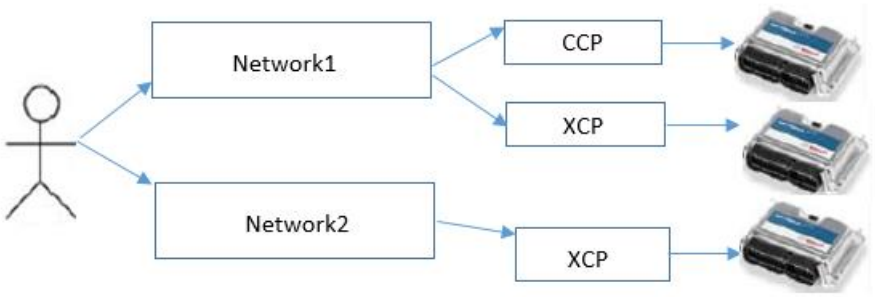


FIGURE 5.15: Multi-Threaded API Access diagram

As shown in the figure 5.15, it might be possible that client wants to connect multiple ECUs on same network and on different network too. The API should be made thread safe while dealing with such use cases.

5.5.2 Solution:

A profile based system has been created which will serve the purpose of hiding the information and handling multiple clients simultaneously. *Network Profile* and *Calibration Profile* are two classes which will hold information about each network and each ECU respectively. *Calibration Profile* class is nothing but a data structure which will hold the A2l data, calibration protocol using by the ECU and other important information. *Network* class holds the information about each network. The *ECUAccessManager* will communicate with each network separately and holds a map for calibration profiles and their corresponding facade objects as shown in the figure 5.16.

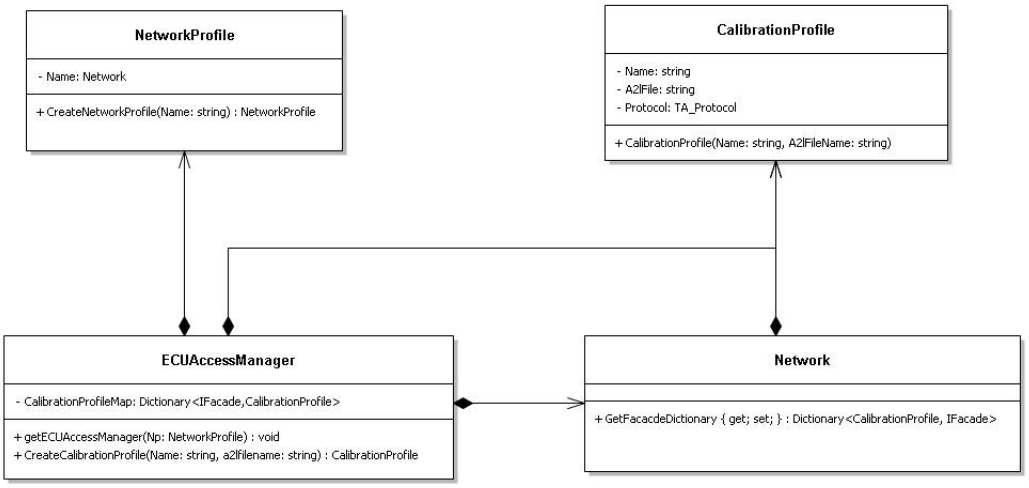


FIGURE 5.16: Multiple Network Class Diagram

- **Network Profile:** This class will hold the network related information for the user.
- **Calibration Profile:** This is the most important class which will make the solution to have multiple client communication possible. This class will hold all the information about each ECU and then mapped to the network class. It contains very important information such as A2lFile, calibration protocol used by the mapped ECU, Hardware Device mapped to the ECU etc.
- **ECUAccessManager:** As stated earlier also, this class acts as a bridge between API and the whole solution. So, this class will create a Network object for each network and holds the reference for future access. This class also holds the information about each calibration profile which has been mapped to the facade objects.
- **Network:** This class holds the map for calibration protocols and the actual facade references. Whenever a request is made with the calibration profile through *ECUAccessManager*, the Network accesses its corresponding facade object and executes the request. Thus with the help of the profile based system, the information from the client can be hid and it also answer the RQ4 by having multiple networks and calibration profiles in parallel.

5.6 SOLID Principles:

Following these principles[16] will help a programmer to build a system which is easy to extend and maintain over time.

"Single responsibility principle"[16] A class should have only one responsibility, so that a change in particular responsibility will not affect or influence other tasks or classes.

"Open Close Principle"[16] The code must be open for extension and closed for modification. It reinforces the idea of polymorphism.

"Liskov substitution principle" "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program."[16]

"Interface segregation principle"[16] It is better to have many different client specific interfaces rather than one general purpose interface.

"Dependency inversion principle"[16] It is a general programming principle to depend on interface rather than concrete classes.

This principle has been used in overall design of the tool. These principles have helped to achieve the goal which have been mentioned in the research questions.

Chapter 6

Implementation

In this chapter discussion about the system design, Parser, usage of tool and validation of tool will be done with the benchmarks set in the beginning of the thesis.

6.1 A2L File Parser

This section will discuss about answer provided for RQ1. As discussed in previous chapter *Regular Expressions* and *ANTLR* has been used to extract information from A2l file.

6.1.1 Regular Expressions

To get the calibration protocol information following regular expressions have been written:

```
@"(\/begin XCP_ON_CAN)(.*?)(\/end XCP_ON_CAN)

@-start of regex
\-Hides special meaning of a character
.-match everything
*?-zero or more times but as few as possible
()-grouping
```

The above stated regular expression will search for the sentence *begin XCP_ON_CAN* in the A2l File. If it is present then the protocol is *XCP_ON_CAN*.

To extract CCP information the solution provided by "CCP Measurement" tool mentioned in the 4.2 has been reused.

6.1.2 ANTLR

The protocol layer information as present in the A2L file is shown in the figure below. This information has to extracted from the A2l file as shown in Appendix B and store it in the data structures. As discussed earlier, there should be a grammar file which should be passed as an input to the tool.

```
/begin PROTOCOL_LAYER

0x0102                      /* XCP protocol layer version */

1000                         /* T1 [ms] */
1000                         /* T2 [ms] */
0                           /* T3 [ms] */
0                           /* T4 [ms] */
0                           /* T5 [ms] */
0                           /* T6 [ms] */
0                           /* T7 [ms] */

8                           /* MAX_CTO */
8                           /* MAX_DTO default for DAQ and STIM */

BYTE_ORDER_MSB_FIRST        /* BYTE_ORDER */
ADDRESS_GRANULARITY_BYTE    /* ADDRESS_GRANULARITY */

OPTIONAL_CMD GET_COMM_MODE_INFO
OPTIONAL_CMD SET_MTA

COMMUNICATION_MODE_SUPPORTED        /* optional modes supported */
BLOCK
SLAVE                   /* Slave Block Mode supported  */
MASTER                  /* Master Block Mode supported */
43   /* MAX_BS */
0     /* MIN_ST */

/end PROTOCOL_LAYER
```

FIGURE 6.1: Protocol Layer Information in A2l File

6.1.2.1 Grammar File

The grammar file which contains information about XCP protocol rules has been written. The grammar to extract information from above snippet will look like:

```
protocollayer:startword='/begin' protoword = 'PROTOCOL_LAYER'
              xcpver=WORDTOKEN
              t1=INTTOKEN
              t2=INTTOKEN
              t3=INTTOKEN
              t4=INTTOKEN
              t5=INTTOKEN
              t6=INTTOKEN
              t7=INTTOKEN
              maxcto=INTTOKEN
              maxdto=INTTOKEN
              byteorder=WORDTOKEN
              (addrsgranbyte = WORDTOKEN)?
              (opt)*
              endword='/end PROTOCOL_LAYER';
```

FIGURE 6.2: Grammar to extract Protocol Layer Information from A2L

```
fragment Digit  : [0-9];
fragment INT : Digit+ ;
fragment Word : ('a'..'z' | 'A'..'Z'|'_'| Digit |'"')+;

WORDTOKEN : Word;
INTTOKEN : INT ;
```

FIGURE 6.3: TOKENS for Grammar

The first word *protocollayer* is the name of rule. The rule name should always be in small case. Then the *startword* and *protoword* is given. Next it is needed to store the XCP version number in a variable. As it can be seen that the version can contain alphabets and numbers both. So, the first line after *begin PROTOCOL_LAYER* should of *WORDTOKEN* type and will be stored in the variable *xcpver*. Similarly, to store the *time* information *INTOKEN* has been written because the value contain only digits. Information after *ADDRESS_GRANULARITY* section is not needed. So a sub rule *opt* has been created which will read all the *WORDTOKENS* after *ADDRESS_GRANULARITY* and will do nothing till it reaches the end line *end PROTOCOL_LAYER*.
The grammar file will run over the A2l file and following files will be generated by the ANTLR tool:

- XCPGrammarBaseListener.cs
- XCPGrammarLexer.cs
- XCPGrammarBaseVisitor.cs
- XCPGrammarParser.cs
- XCPGrammarVisitor.cs

The class diagram shown in the figure 6.4 has shown how these classes can be used to store the information in the data structures.

6.1.2.2 XCP Parser Class Diagram

As shown in the figure 6.4, user will call the method *ParseXCP* which is defined in the class *A2lProtocolParser*. In this method user will call *protocollayer* method which is defined in the class *XCPGrammarParser* automatically by the ANTLR tool. Remember user have set the rule name as *protocollayer* in the grammar file. Next user will create an object of *XCPParserVisitor* class in the method *ParseXCP*. This object will call *VisitProtocollayer* method. This method has argument as the extracted data in string format. From there the information will be stored in the *XCP_Calibration* class. user can simply call the *GetProtocolLayer* property to get the information present in the a2l file.
Similarly, to extract the CCP related information user have used the grammar file defined by the *CCP_Measurement* tool 4.2.

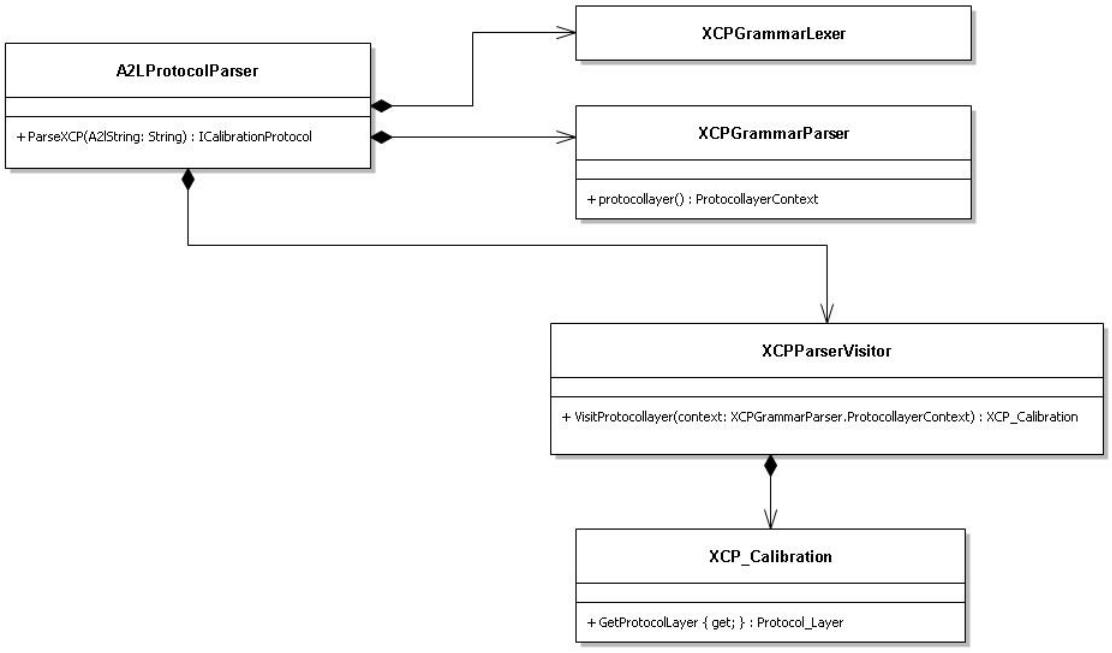


FIGURE 6.4: XCP parser Class Diagram

6.2 User Interface

Now the discussion on User Interface which have been created for the tool will be done. The main objective was to keep the UI as simple as possible so that it should take very less time to upload. So, basically there were three requirements before creating the UI. These were:

- Quick to load
- Easy to modify
- User friendly

At the end of this section evaluation will be done to check whether all the requirements have been fulfilled or not.

6.2.1 UI Class Diagram

The class diagram of the UI is shown in the figure 6.5. From the class diagram it can be seen that design has been kept as simple as possible. There is no cyclic dependency between classes. To optimize further creating object of each classes in another class has been taken care of. It has also be made sure that there is only one way by which the UI can connect to the entire network. This has been done by creating *APIConnector* class. So, view is totally independent from model as required in the MVC design. For e.g. suppose in future if the UI is extended on XCP on Flexray then no change has to be done on the UI.
Secondly, by making *APIConnector* as a singleton class any possible ambiguity while accessing the network has been omitted.

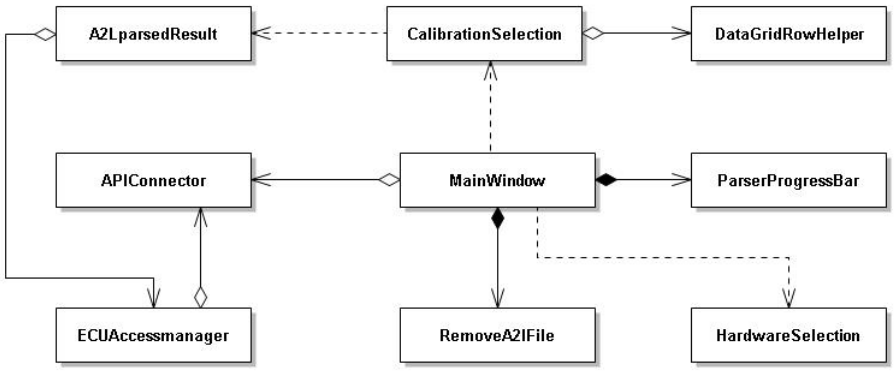


FIGURE 6.5: User Interface Class Diagram

6.2.2 Validating User Interface

Now check whether the requirements discussed in the beginning of this section have met or not. The UI hardly takes any time to load. Secondly, in case of modification the developer has to just add a new class and create instance of the class in *MainWindow*

class thus making it easy to modify.
Thirdly, feedback from many users have been taken and asked them whether the UI is user friendly or not. Provided suggestions have been integrated to make it more user friendly. Also *Windows Presentation Framework (WPF)* has been used instead of traditional *Windows Form* which has improved the look of the UI. WPF has also been used keeping in mind about further extension of the tool in future. WPF gives more modularity hence it will be easy to extend it.
Thus by using proper design the requirements imposed before designing the UI have been met. The overview of UI has been shown in the Figure 6.6. The detailed usage of UI will be explained in later section.

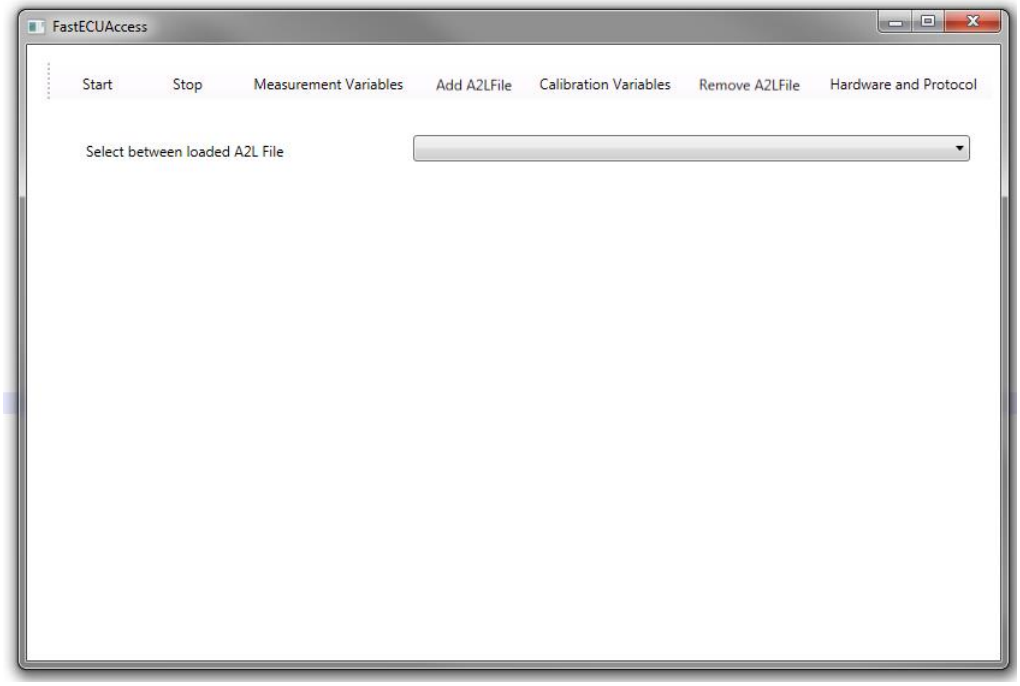


FIGURE 6.6: User Interface of the Tool

6.3 Architecture of the Tool

In this section the implementation of the architecture will be discussed. Again, before designing the architecture there were following requirements:

- Architecture should be easy to extend
- Must be reusable

At the end of this section evaluation will be done to check whether all the requirements have been fulfilled or not. Discussion on overall class diagram of the tool will also be done.

6.3.1 Overall Class Diagram

The picture shown in figure 6.7 shows the overall class diagram of the tool. So, basically it has shown how different classes are dependent on each other. Each class has a single responsibility, which has fulfilled the first point of SOLID Principle. To achieve open/close principle the solution have implemented many interfaces. So in future if the code needs to be extended then only the interface needs to be modified. Similarly, the SOLID principles have been followed in the design.

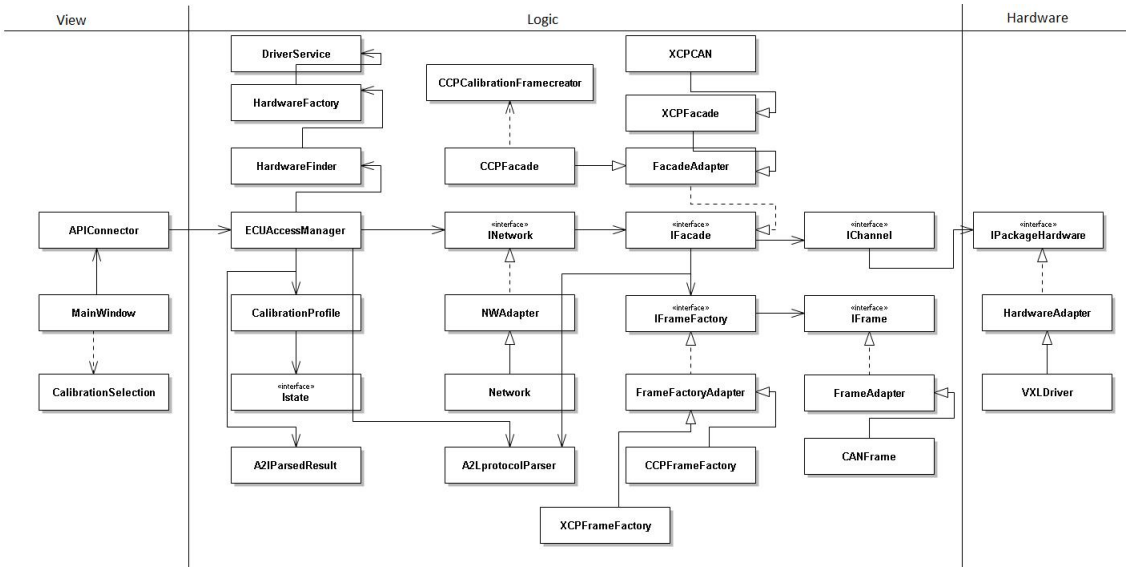


FIGURE 6.7: Overall Class Diagram

As shown in the figure 6.7, there is a single point entry to the network and that is possible via *ECUAccessManager* class. This class then create *Network Profiles* for each network. As discussed earlier, each ECU will be mapped to a unique *CalibrationProfile*. Each *Network* object is mapped to the corresponding *Facade* object. Information such as which hardware device is mapped to the corresponding ECU, state of the ECU will be fetched from *CalibrationProfile* class and given to the *Facade* class. Depending on the calibration protocol in the a2l file the mapped *Facade* object will be called. *CCPFacade* or *XCPFacade* will communicate to hardware via *Channel* class. Once the message is received in the hardware then an event will raised. This event is then handled in the *Channel* class. After filtering message according to the message identifier a new

event will be raised which is handled in *CCPFacade* and *XCPFacade* depending on the facade objects. After processing the message, "Facade" will return this frame to *ECUAccessManager* class and then to *APIConnector* class which then update the User Interface. For logging the intermediate information a very powerful tool *Log4Net* has been used. The tool is developed by *Apache.org*. This design has just imported the dlls in the project and called their methods to log the information.

6.3.2 Validating Architecture

As SOLID principles have been strictly followed in this design, so the architecture can be easily extended. After analyzing each class if it has been figured that the class can be extended in future then the class has been mapped to an interface. So, due to modularity of architecture it is easy to maintain.
So, all the requirements which have been imposed before designing the architecture have been successfully met.

6.4 Validation

Finally validation of results will be done. The solution can be validated using several approach. Some of them are discussed below:

- **Validating using GUI**:
  In this approach calibration variables of an ECU will be loaded and their calibration will be performed.The calibration variables of multiple ECUs will also be loaded and performance will be evaluated.
- **Integrating into automation process**:
  In this approach calibration variables of an ECU will be loaded to calibrate them. The calibration variables of multiple ECUs will also be loaded and performance will be evaluated.

6.4.1 Calibration using GUI

6.4.1.1 Single ECU

Step by step discussion on how to calibrate variables using GUI will be done in this section. An ECU from BMW motor cycle which supports CCP has been used for testing. Calibration of internal parameters of this ECU will be done.

After opening the tool, the very first step is to load the a2l file of the ECU. This can be done by clicking on "Add A2l File" button as shown in the figure 6.8.

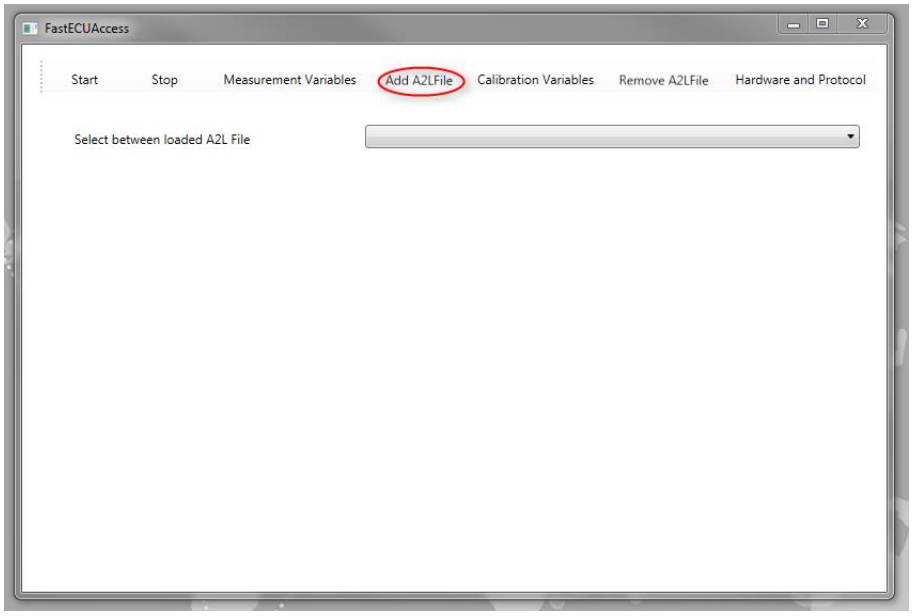


FIGURE 6.8: Add A2l File in FastEcuAccess Tool

A new *OpenFileDialog* window will open. Select the a2l file from there. The progress of reading the file will be shown by a parser progress bar as shown in the figure 6.9.

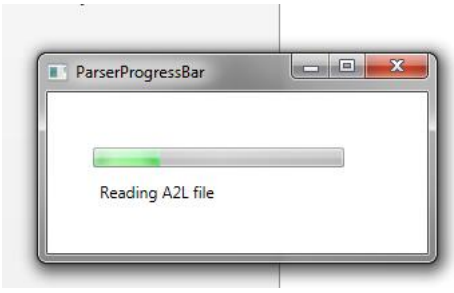


FIGURE 6.9: Progress bar for reading A2l File

Once the file is loaded, then click on "Calibration Variables" button as shown in the figure 6.10 to get the list of all the calibration variables present in the a2l file.

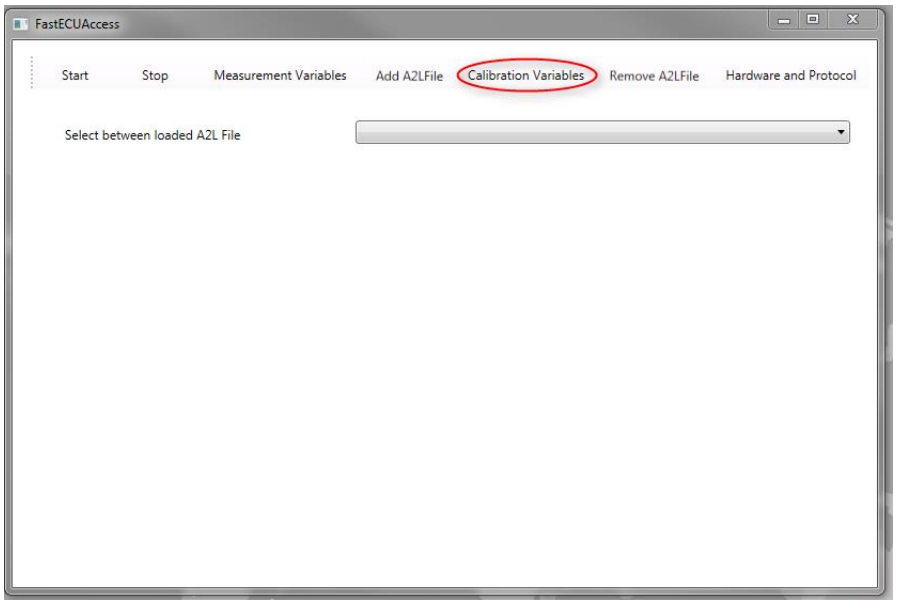


FIGURE 6.10: To get the Calibration Variables of ECU

A new window will open as shown in the figure 6.11. This window contains the list of all the calibration variables of the ECU. User has to select the calibration variables from the list. A search box has also been added for directly searching variables from the list. After selecting the variables user has to click on *OK* button. As shown in the figure 6.12, all the selected variables then added into the main window with few information about them. The information includes the current physical value, unit, current internal value and two text boxes. In the first text box, user will enter the new value which they wants to write in the ECU. The second text box is for calibrating *Curves* and *Maps*. As shown in the figure 6.12, variable *UEGO_idxUpNorm3_CUR* has values 1,3,7,10,13. If user wants to change 4th value then in the "Offset" text box they have to enter "3" (Zero based Indexing).

For e.g. suppose user wants to change the max permissible speed for verification test (Calibration Variable: ATS_TstDemMaxEngN_C) to 2000 rpm. Also user wants to change the 4th value of *UEGO_idxUpNorm3_CUR* to 12. User has to enter the values as shown in the figure 6.13 and click on "OK" button of the corresponding to the row in which the variable is present.

The new value will be updated in the ECU RAM and to validate it has been read again. As it can be seen in the figure 6.14, the new values are updated in the UI.

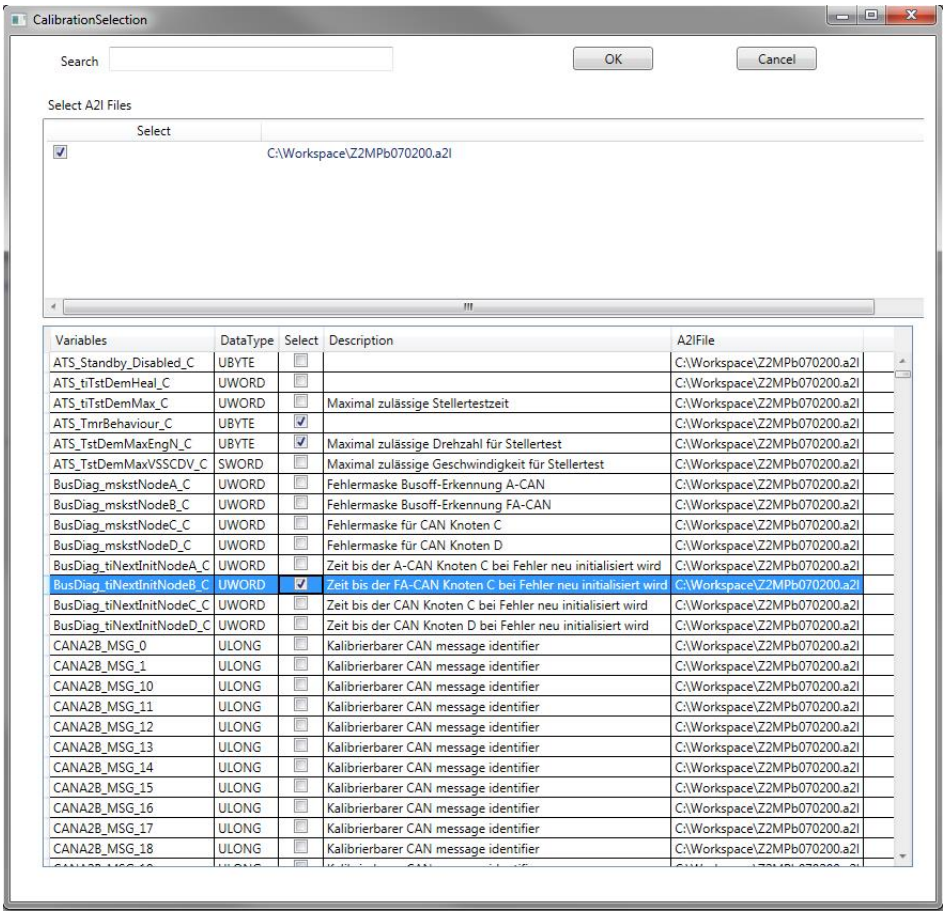


FIGURE 6.11: List of Calibration Variables in the BMW ECU

FIGURE 6.12: Selected Calibration Variables added into Main Window

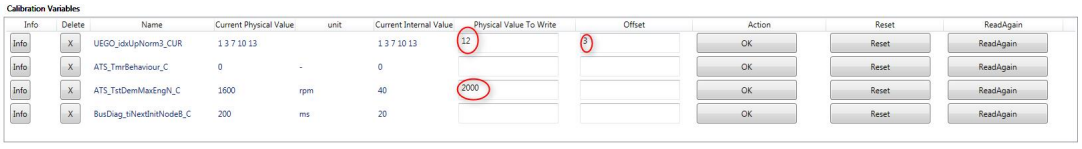
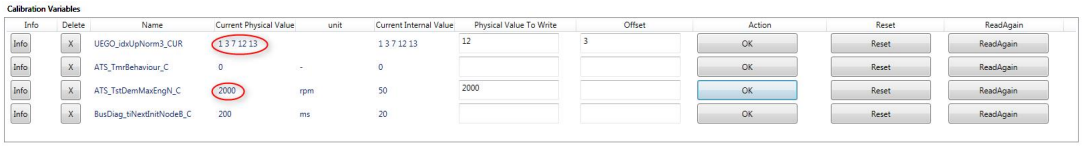


FIGURE 6.13: Value to be written for calibration variables



Calibration Variables

Info	Delete	Name	Current Physical Value	unit	Current Internal Value	Physical Value To Write	Offset	Action	Reset	ReadAgain
Info	X	UEGO_idxUpNorm3_CUR	1 3 7 12 13		1 3 7 12 13	12	3	OK	Reset	ReadAgain
Info	X	ATS_TmrBehaviour_C	0	-	0			OK	Reset	ReadAgain
Info	X	ATS_TstDemMaxEngN_C	2000	rpm	50	2000		OK	Reset	ReadAgain
Info	X	BusDiag_tiNextInitNodeB_C	200	ms	20			OK	Reset	ReadAgain

FIGURE 6.14: Updated Values of Calibration Variables

So, calibration of the variable of BMW ECU has been successfully done in very few steps. As shown in figure 6.14, there are few extra buttons which have been added too in the main window. *Reset* button as name suggests will reset all the changes which have been made to that variable and set it to the original value i.e. the value which is in ROM. *ReadAgain* button will read the value of the variable again from the ECU RAM. *Info* button will give some more information about the variable. This includes name, Data Type, minimum value and maximum value of the variable.

6.4.1.2 Multiple ECU

In this section one more ECU will be added which supports XCP on CAN in the system. So now the system have one BMW motorrad ECU (supporting CCP) and another ECU which supports XCP on CAN. User will calibrate variables of both the ECUs connected in parallel with very few change in steps. Further in this section ECU1 refers to *BMW ECU* and *ECU2* refers to the second ECU. The a2l file corresponding to ECU1 is *C:\Workspace\Z2MPb070200.a2l* and a2l file corresponding to ECU2 is *C:\Workspace\VC1CP102HMC01_11H0.a2l*.
Firstly, user have to load both the a2l files corresponding to the ECUs. As it can be seen in the figure 6.15, there are two a2l files shown in the window. User can select the a2l file from which they want to select the calibration variables. In this case suppose user have chosen the 1st two variables of both the a2l files as shown in the figure 6.15 and 6.16.

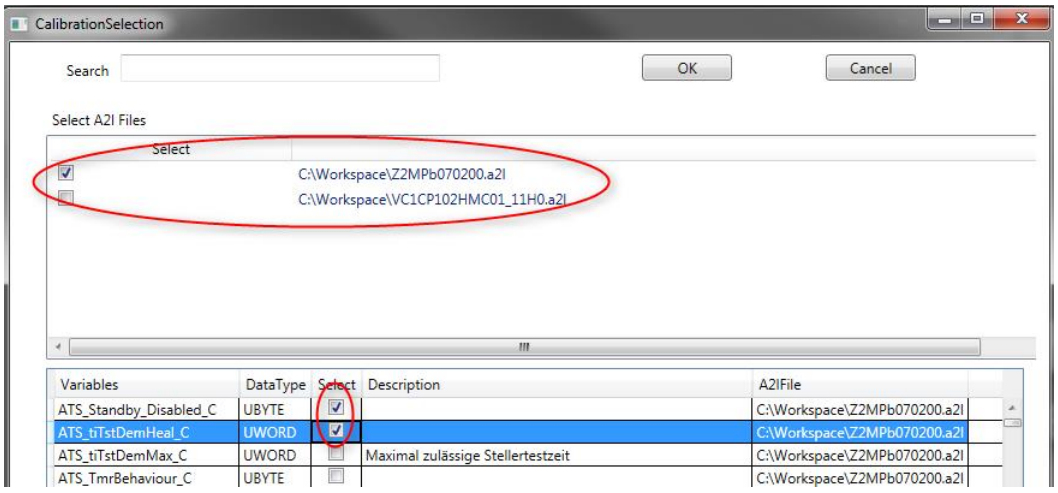


FIGURE 6.15: List of Calibration Variables from ECU1

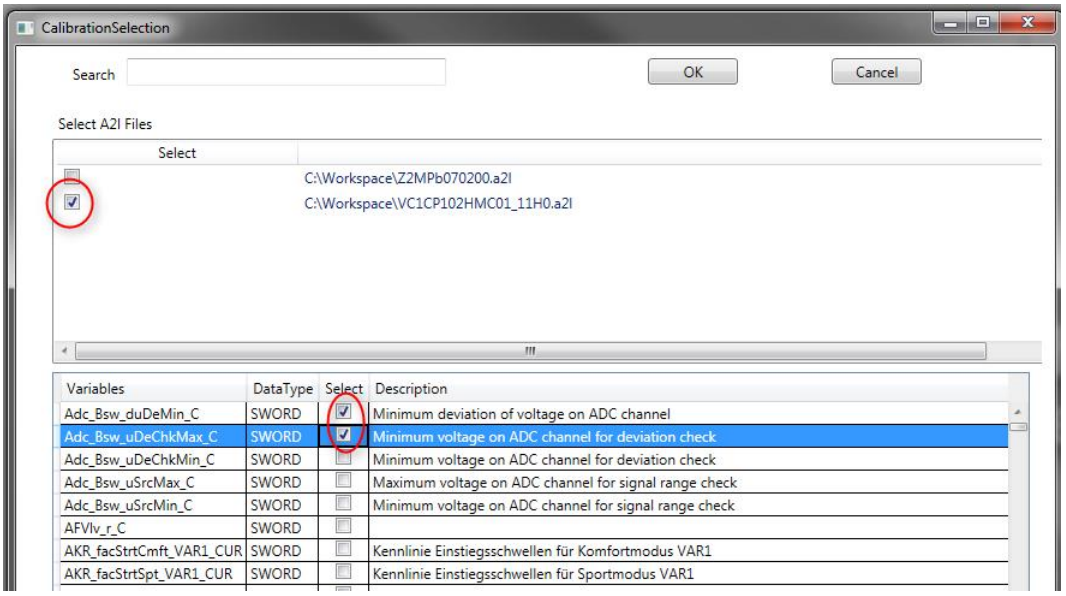


FIGURE 6.16: List of Calibration Variables from ECU2

As shown in the figure 6.17, variables "ATS_Standby_Disabled_C" and "ATS_tiTstDemHeal_C" are from ECU1 and remaining two variables are from ECU2. Suppose user wants to change "ATS_Standby_Disabled_C" to 78 and "Adc_Bsw_duDeMin_C" to 300 then user will give the value accordingly in the text boxes and click on OK button as discussed in previous section. The new value is updated and can be seen in figure 6.18.

Calibration Variables

Info	Delete	Name	Current Physical Value	unit	Current Internal Value	Physical Value To Write	Offset	Action
Info	X	ATS_Standby_Disabled_C	1	-	1	78		OK
Info	X	ATS_tiTstDemHeal_C	0	s	0			OK
Info	X	Adc_Bsw_duDeMin_C	200	mV	1000	300		OK
Info	X	Adc_Bsw_uDeChkMax_C	3100	mV	15500			OK

FIGURE 6.17: Selected Variables from ECU1 and ECU2

Calibration Variables

Info	Delete	Name	Current Physical Value	unit	Current Internal Value	Physical Value To Write
Info	X	ATS_Standby_Disabled_C	78	-	78	78
Info	X	ATS_tiTstDemHeal_C	0	s	0	
Info	X	Adc_Bsw_duDeMin_C	300	mV	1500	300
Info	X	Adc_Bsw_uDeChkMax_C	3100	mV	15500	

FIGURE 6.18: Updated values of Calibration Variables from ECU1 and ECU2

So, it can be seen that in case of multiple ECUs connected in parallel the variables from all the ECUs can be easily. This answered the RQ4.

6.4.2 Integration

Integration of the API into an internally developed tool named *Automated Integration Test* tool has also been done. This tool is used internally to automate the entire process of testing. The tool will take python script as an input and generate the results and shows which test was failed and vice versa. Although this tool has many other features user will stick to calibration of internal parameters of ECU on CCP and XCP on CAN. Before integrating the solution this tool uses INCA to calibrate the variables.
The integration has been done in such a way that the same python script can be used for testing using INCA as well as testing using this solution (called as *ECUAccess*). A simple python script for calibrating variables as shown in Appendix C has been written. The script is just calibrating two variables and measuring one variable on CCP. To validate the performance user have run the same script using INCA and this solution. The time taken by both the process has been noted down. At last time taken by both the solutions to perform testing have been compared. The benchmark to validate the solution was that it should take at least half of the time as taken by INCA.

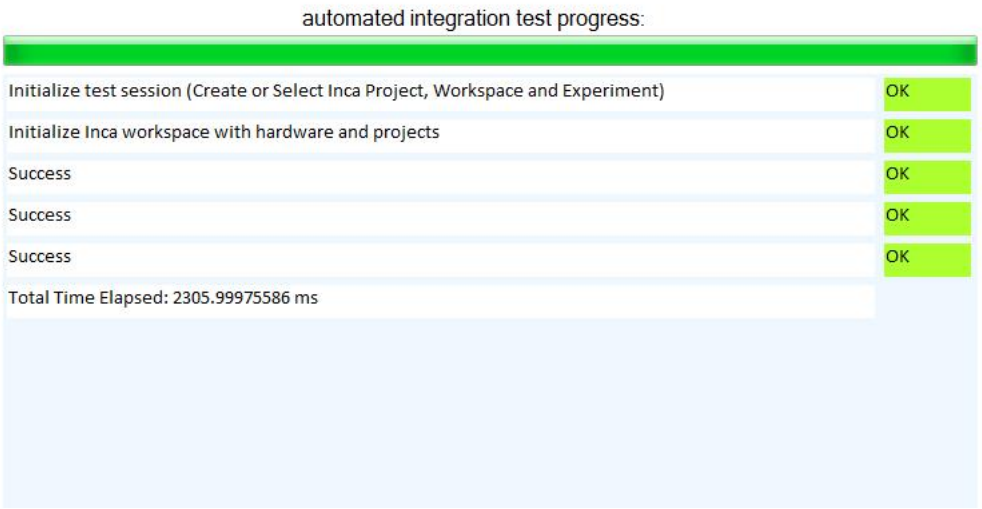


FIGURE 6.19: Integration Test Result using INCA

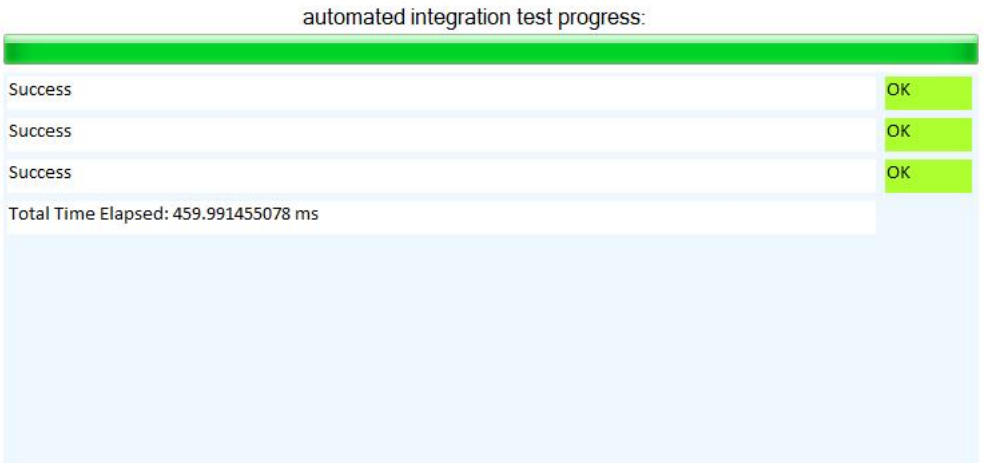


FIGURE 6.20: Integration Test Result using ECUAccess

As shown in the figure 6.19, the test script as mentioned in Appendix C when run with INCA takes 2.305 seconds to complete the test while when run with ECUAccess the tests finished in 0.459 seconds as shown in the figure 6.20. Hence this solution is taking almost 80 percent less time as taken by existing solution. Hence validation of the solution with respect to time, single ECU communication and multiple ECU communication has been successfully done.

Chapter 7

Future Work

This chapter will discuss about the future works which can be done with respect to the tool. This will give an insight to the reader while referring the thesis.

7.1 Protocol

Current solution supports XCP only on CAN. In future this can be extended to different transport layers such as Flexray, Ethernet, LIN etc.
Secondly, the current solution can be further extended to have ETK support in it.

7.2 Functionality

This solution supports calibration using CCP and XCP on CAN. In future this can be extended to do measurement of variables using XCP.
Secondly, flashing the ECU is possible via CCP and XCP. But this has not been implemented in this solution. This feature can be added in the future.

7.3 Hardware

The current solution can run on Vector Hardware and Micro Lab Car device. The solution can be further extended for different hardware. This can be easily done by changing the *Ihardware* interface.

Chapter 8

Conclusion

In a nutshell, from this thesis a UI and API have been developed which can be used in test automation system to calibrate the internal ECU parameters present in the a2l file. "ANTLR" has been used to parse the information from a2l file and "log4net" to log the information. With the help of these open source tools the performance of tool has been improved significantly. The current solution which was being used for calibration was very slow and hence used to slow the entire automation process. The solution provided by this thesis have successfully shown that the solution is almost 5 times faster than the existing solution. Hence this has improved the performance of automation system with respect to time. The solution have been also successfully tested with multiple ECUs connected in parallel. With the help of this feature the Master/Slave ECU can be tested easily in the automation system. The discussion about the architecture of the tool has also been done. It has shown how this approach will make the architecture easy to use, maintain and extend. For designing our UI MVC pattern has been used hence making the UI easy to load and independent from the complex logic of the tool. The future works have been also mentioned clearly in the report.

Appendix A

A2L Calibration Variable Information

Calibration Variable

```
/begin CHARACTERISTIC

Adc_Bsw_duDeMin_C
"Minimum deviation of voltage on ADC channel"
VALUE
0x95803D6
DefaultValueRecordLayout_sint16
65535.0
RB_RBA_Adc_Bsw_CompuMethods_Adc_Bsw_Deviation_comp
-32768.0
32767.0

FORMAT "%5.0"


/end CHARACTERISTIC


/begin CHARACTERISTIC

Adc_Bsw_uDeChkMax_C
"Minimum voltage on ADC channel for deviation check"
```

```
VALUE
0x95803D4
DefaultValueRecordLayout_sint16
65535.0
RB_RBA_Adc_Bsw_CompuMethods_Adc_Bsw_Value_comp
-32768.0
32767.0

FORMAT "%5.0"


/end CHARACTERISTIC


/begin CHARACTERISTIC

Adc_Bsw_uDeChkMin_C
"Minimum voltage on ADC channel for deviation check"
VALUE
0x95803D2
DefaultValueRecordLayout_sint16
65535.0
RB_RBA_Adc_Bsw_CompuMethods_Adc_Bsw_Value_comp
-32768.0
32767.0

FORMAT "%5.0"


/end CHARACTERISTIC


/begin CHARACTERISTIC

Adc_Bsw_uSrcMax_C
"Maximum voltage on ADC channel for signal range check"
VALUE
0x95803D0
DefaultValueRecordLayout_sint16
```

```
65535.0
RB_RBA_Adc_Bsw_CompuMethods_Adc_Bsw_Value_comp
-32768.0
32767.0

FORMAT "%5.0"



/end CHARACTERISTIC



/begin CHARACTERISTIC

Adc_Bsw_uSrcMin_C
"Minimum voltage on ADC channel for signal range check"
VALUE
0x95803CE
DefaultValueRecordLayout_sint16
65535.0
RB_RBA_Adc_Bsw_CompuMethods_Adc_Bsw_Value_comp
-32768.0
32767.0

FORMAT "%5.0"



/end CHARACTERISTIC
```

Appendix B

A2L XCP Information

XCP Protocol

```
/begin XCP_ON_CAN

/************************ start of CAN *********************/

0x0102 /* XCP on CAN version */
CAN_ID_MASTER 0x7F0         /* CMD/STIM CAN-ID */
/* master -> slave */
CAN_ID_SLAVE 0x7F1          /* RES/ERR/EV/SERV/DAQ CAN-ID */
/* slave -> master */

BAUDRATE 500000            /* BAUDRATE [Hz] */
SAMPLE_POINT 70             /* sample point */
/* [% complete bit time] */
SAMPLE_RATE SINGLE          /* 1 sample per bit */
BTL_CYCLES 10               /* BTL_CYCLES */
/* [slots per bit time] */
SJW 1                       /* length synchr. segment */
/* [BTL_CYCLES] */
SYNC_EDGE SINGLE            /* on falling edge only */
MAX_BUS_LOAD  88            /* maximum available bus */

/************************ end of CAN *********************/

/*************** start of PROTOCOL_LAYER ******************/
```

```
/begin PROTOCOL_LAYER

0x0102                      /* XCP protocol layer version */

1000                         /* T1 [ms] */
1000                         /* T2 [ms] */
0                           /* T3 [ms] */
0                           /* T4 [ms] */
0                           /* T5 [ms] */
0                           /* T6 [ms] */
0                           /* T7 [ms] */

8                           /* MAX_CTO */
8                           /* MAX_DTO default for DAQ and STIM */

BYTE_ORDER_MSB_FIRST        /* BYTE_ORDER */
ADDRESS_GRANULARITY_BYTE    /* ADDRESS_GRANULARITY */

OPTIONAL_CMD GET_COMM_MODE_INFO
OPTIONAL_CMD SET_MTA
OPTIONAL_CMD UPLOAD
OPTIONAL_CMD SHORT_UPLOAD
OPTIONAL_CMD BUILD_CHECKSUM
OPTIONAL_CMD DOWNLOAD
OPTIONAL_CMD DOWNLOAD_NEXT
OPTIONAL_CMD DOWNLOAD_MAX
OPTIONAL_CMD SET_CAL_PAGE
OPTIONAL_CMD GET_CAL_PAGE
OPTIONAL_CMD COPY_CAL_PAGE
OPTIONAL_CMD CLEAR_DAQ_LIST
OPTIONAL_CMD SET_DAQ_PTR
OPTIONAL_CMD WRITE_DAQ
OPTIONAL_CMD SET_DAQ_LIST_MODE
OPTIONAL_CMD GET_DAQ_LIST_MODE
OPTIONAL_CMD START_STOP_DAQ_LIST
OPTIONAL_CMD START_STOP_SYNCH
OPTIONAL_CMD FREE_DAQ
OPTIONAL_CMD ALLOC_DAQ
```

```
OPTIONAL_CMD ALLOC_ODT
OPTIONAL_CMD ALLOC_ODT_ENTRY

COMMUNICATION_MODE_SUPPORTED        /* optional modes supported */
BLOCK
SLAVE                   /* Slave Block Mode supported  */
MASTER                  /* Master Block Mode supported */
43   /* MAX_BS */
0     /* MIN_ST */

/end PROTOCOL_LAYER

/***************** end of PROTOCOL_LAYER *******************/

/********************** start of DAQ **********************/

/begin DAQ                            /* DAQ supported, at MODULE*/
DYNAMIC                             /* DAQ_CONFIG_TYPE */
65535                               /* MAX_DAQ */
2                                   /* MAX_EVENT_CHANNEL */
0                                   /* MIN_DAQ */
OPTIMISATION_TYPE_DEFAULT           /* OPTIMISATION_TYPE */
ADDRESS_EXTENSION_FREE              /* ADDRESS_EXTENSION */
IDENTIFICATION_FIELD_TYPE_ABSOLUTE  /* IDENTIFICATION_FIELD */
GRANULARITY_ODT_ENTRY_SIZE_DAQ_BYTE /* GRANULARITY_ODT_ENTRY_SIZE_DAQ */
255                                 /* MAX_ODT_ENTRY_SIZE_DAQ */
OVERLOAD_INDICATION_EVENT           /* OVERLOAD_INDICATION */
PRESCALER_SUPPORTED
/begin DAQ_MEMORY_CONSUMPTION
5497      /* DAQ_MEMORY_LIMIT    : in Elements[AG] */
54          /* DAQ_SIZE             : Anzahl Elements[AG] pro DAQ-Liste */
12          /* ODT_SIZE             : Anzahl Elements[AG] pro ODT */
5          /* ODT_ENTRY_SIZE       : Anzahl Elements[AG] pro ODT_Entry */
2          /* ODT_DAQ_BUFFER_ELEMENT_SIZE : Anzahl Payload-Elements[AG]*Faktor = size
3          /* ODT_STIM_BUFFER_ELEMENT_SIZE: Anzahl Payload-Elements[AG]*Faktor = size
/end DAQ_MEMORY_CONSUMPTION

/************************ start of EVENT ************************/
```

```
/begin EVENT                        /* EVENT               */
"10ms time synchronous"           /* EVENT_CHANNEL_NAME       */
"10_ms"                           /* EVENT_CHANNEL_SHORT_NAME */
0                                 /* EVENT_CHANNEL_NUMBER     */
DAQ                               /* EVENT_CHANNEL_TYPE     */
1                                 /* MAX_DAQ_LIST */
10                                 /* EVENT_CHANNEL_TIME_CYCLE */
6                                 /* EVENT_CHANNEL_TIME_UNIT */
6                                 /* EVENT_CHANNEL_PRIORITY */
/begin MIN_CYCLE_TIME             /* Configuration with 0-0 not allowed */
14                                /* EVENT_CHANNEL_TIME_CYCLE */
6                                /* EVENT_CHANNEL_TIME_UNIT */
/end MIN_CYCLE_TIME
/end EVENT

/begin EVENT                        /* EVENT               */
"100ms time synchronous"           /* EVENT_CHANNEL_NAME       */
"100_ms"                           /* EVENT_CHANNEL_SHORT_NAME */
1                                 /* EVENT_CHANNEL_NUMBER     */
DAQ                               /* EVENT_CHANNEL_TYPE     */
1                                 /* MAX_DAQ_LIST */
100                                 /* EVENT_CHANNEL_TIME_CYCLE */
6                                 /* EVENT_CHANNEL_TIME_UNIT */
5                                 /* EVENT_CHANNEL_PRIORITY */
/end EVENT

/******************* end of EVENT *******************/

/end DAQ

/******************** end of DAQ *************************/

TRANSPORT_LAYER_INSTANCE "Calibration CAN (CAN_1)"
/end XCP_ON_CAN
```

Appendix C

Integration Test

Python Script

```
from timeit import default_timer as timer
start = timer()
SwitchT15(True)
AddIncaLabel("Adc_Bsw_uSrcMax_C")
AddIncaLabel("Adc_Bsw_duDeMin_C")
AddIncaLabel("Adc_Bsw_uDeChkMax_C")


DownloadWorkpageAndReferencepage()
SwitchToWorkPage()
SetValue("Adc_Bsw_duDeMin_C",200)
value = GetValue("Adc_Bsw_duDeMin_C")
if(value == str(value)):
LogSuccess("Success")
else:
LogFailure("Failed")

SetValue("Adc_Bsw_uDeChkMax_C",1500)
value = GetValue("Adc_Bsw_uDeChkMax_C")
if(value == str(value)):
LogSuccess("Success")
else:
LogFailure("Failed")
```

```
SetValue("Adc_Bsw_uSrcMax_C",4000)
value = GetValue("Adc_Bsw_uSrcMax_C")
if(value == str(value)):
LogSuccess("Success")
else:
LogFailure("Failed")


SwitchT15(False)
end = timer()
value = (end - start) * 1000
Log("Total Time Elapsed: " + str(value) + " ms")
# ==================================== EOF====================
```

Appendix D

XCP Commands Response

CONNECT

Positive Response:

Table D.1: XCP Connect Command Positive Response

Position	Type	Description
0	BYTE	Packet ID: 0xFF
1	BYTE	RESOURCE
2	BYTE	RESOURCE
3	BYTE	COMM_MODE_BASIC
4	BYTE	MAX_CTO, Maximum CTO size [BYTE]
5	BYTE	MAX_DTO, Maximum DTO size [BYTE]
6	BYTE	XCP Protocol Layer Version Number (most significant byte only)
7	BYTE	XCP Transport Layer Version Number (most significant byte only)

Source:[6]

Negative Response:

Table D.2: XCP Connect Command Negative Response

Position	Type	Description
0	BYTE	PACKET ID = 0xFE
1	BYTE	Error Code

Source:[6]

XCP GET_STATUS

Positive Response:

Table D.3: XCP GET_STATUS Command Positive Response

Position	Type	Description
0	BYTE	Packet ID: 0xFF
1	BYTE	Current session status
2	BYTE	Current resource protection status
3	BYTE	Reserved
4	BYTE	Session configuration id

Source:[6]

Negative Response:

Table D.4: XCP GET_STATUS Command Negative Response

Position	Type	Description
0	BYTE	PACKET ID = 0xFE
1	BYTE	Error Code

Source:[6]

UPLOAD

Positive Response:

Table D.5: XCP UPLOAD Command Positive Response

Position	Type	Description
0	BYTE	Packet ID: 0xFF
1	BYTE	Used only for alignment
2	BYTE	1st Data Element
3	BYTE	2nd Data Element
7	BYTE	nth Data Element

Source:[6]

Negative Response:

Table D.6: XCP UPLOAD Command Negative Response

Position	Type	Description
0	BYTE	PACKET ID = 0xFE
1	BYTE	Error Code

Source:[6]

DOWNLOAD

Positive Response:

Table D.7: XCP DOWNLOAD Command Positive Response

Position	Type	Description
0	BYTE	Packet ID: 0xFF

Source:[6]

Negative Response:

Table D.8: XCP DOWNLOAD Command Negative Response

Position	Type	Description
0	BYTE	PACKET ID = 0xFE
1	BYTE	Error Code

Source:[6]

SET_MTA

Positive Response:

Table D.9: XCP SET_MTA Command Positive Response

Position	Type	Description
0	BYTE	Packet ID: 0xFF

Source:[6]

Negative Response:

Table D.10: XCP SET_MTA Command Negative Response

Position	Type	Description
0	BYTE	PACKET ID = 0xFE
1	BYTE	Error Code

Source:[6]

GET_CAL_PAGE

Positive Response:

Table D.11: XCP GET_CAL_PAGE Command Positive Response

Position	Type	Description
0	BYTE	Packet ID: 0xFF
1	BYTE	Reserved
2	BYTE	Reserved
3	BYTE	Page Number

Source:[6]

Negative Response:

Table D.12: XCP GET_CAL_PAGE Command Negative Response

Position	Type	Description
0	BYTE	PACKET ID = 0xFE
1	BYTE	Error Code

Source:[6]

SET_CAL_PAGE

Positive Response:

Table D.13: XCP SET_CAL_PAGE Command Positive Response

Position	Type	Description
0	BYTE	Packet ID: 0xFF
1	BYTE	Reserved
2	BYTE	Reserved
3	BYTE	Page Number

Source:[6]

Negative Response:

Table D.14: XCP SET_CAL_PAGE Command Negative Response

Position	Type	Description
0	BYTE	PACKET ID = 0xFE
1	BYTE	Error Code

Source:[6]

Bibliography

[1] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *Software Engineering, IEEE Transactions on*, 39(5):658–683, May 2013. ISSN 0098-5589. doi: 10.1109/TSE.2012.64. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6311410&isnumber=6509864.

[2] *ASAM-MCD2*, Accesed: 24-October 2016. URL https://wiki.asam.net/display/STANDARDS/ASAM+MCD-2+MC.

[3] CANcaseXL - Manual Accesed: 24-October 2016. URL https://vector.com/portal/medien/cmc/manuals/CANcaseXL_Manual_DE.pdf.

[4] Introduction to CCP Accesed: 24-October 2016. URL http://vector.com/portal/medien/cmc/application_notes/AN-AMC-1-102_Introduction_to_CCP.pdf.

[5] XCP-Reference Book Accesed: 24-October 2016. URL http://vector.com/portal/medien/solutions_for/xcp/XCP_ReferenceBook_V1.0_EN.pdf.

[6] XCP -Part 2- Protocol Layer Specification -1.0. Accesed: 24-October 2016. URL .

[7] XCP Basics Accesed: 24-October 2016. URL https://vector.com/portal/medien/solutions_for/xcp/Vector_XCP_Basics_EN.pdf.

[8] ES581 – USB CAN Bus Interface Accesed: 24-October 2016. URL http://www.etas.com/en/products/es581.php.

[9] ES600 – USB CAN Bus Interface Accesed: 24-October 2016. URL http://www.etas.com/en/products/compact_es600_measurement_modules.php.

[10] E. Freeman, E. Robson, B. Bates, and K. Sierra. *Head First Design Patterns*. Head First. O'Reilly Media, 2004. ISBN 9781449331498.

[11] msdn. *Quality Attributes*, Accesed: 24-October 2016. URL https://msdn.microsoft.com/en-us/library/ee658094.aspx.

[12] Paul Hsieh. Programming optimization. Accesed: 24-October 2016. URL http://www.azillionmonkeys.com/qed/optimize.html.

[13] msdn. Thread pools. Accesed: 24-October 2016. URL https://msdn.microsoft.com/en-us/library/windows/desktop/ms686760(v=vs.85).aspx.

[14] James Michael Hare. C#: System.collections.concurrent.concurrentqueue vs. queue. Accesed: 24-October 2016. URL http://geekswithblogs.net/BlackRabbitCoder/archive/2010/06/07/c-system.collections.concurrent.concurrentqueue-vs.-queue.aspx.

[15] B. McLaughlin, G. Pollice, and D. West. *Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D*. Head first series. O'Reilly Media, 2006. ISBN 9780596008673.

[16] Robert C. Martin. Design principles and design patterns. Accesed: 24-October 2016. URL http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf.

[17] P.M. Heathcote. *'A' Level Computing*. A Level Computing Series. Payne-Gallway Publishers, 2005. ISBN 9781903112212.

[18] Apache. Log4j. Accesed: 24-October 2016. URL http://logging.apache.org/log4j/2.x/.

[19] Adam Brown. Application logging: What, when, how. Accesed: 24-October 2016. URL https://dzone.com/articles/application-logging-what-when.

[20] Robert Bosch GmbH. Can specification. 2.0, Accesed: 24-October 2016 1991. URL http://www.kvaser.com/software/7330130980914/V1/can2spec.pdf.

[21] H. Kleinknecht. Ccp specification. 2.1, 27-December 1999. URL https://automotivetechis.files.wordpress.com/2012/06/ccp211.pdf.

[22] Terence Parr. *The Definitive ANTLR 4 Reference*. Oreilly and Associate Series. Pragmatic Bookshelf, 2013. ISBN 9781934356999.

[23] Paul Houle. The multiton design pattern. Accesed: 24-October 2016. URL http://gen5.info/q/2008/07/25/the-multiton-design-pattern/.

[24] Alexandr Shvets. Design patterns. Accesed: 24-October 2016. URL https://sourcemaking.com/design_patterns.

[25] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. An Alan R. Apt book. Prentice Hall, 1996. ISBN 9780131829572.

[26] M.P.P. Team. *Microsoft® Application Architecture Guide*. Microsoft Press, 2009. ISBN 9780735642799.

[27] Tom Dalling. Mvc design pattern explained. Accesed: 24-October 2016. URL http://www.tomdalling.com/blog/software-design/model-view-controller-explained/.

[28] G. Fairbanks and D. Garlan. *Just Enough Software Architecture: A Risk-Driven Approach*. Marshall & Brainerd, 2010. ISBN 9780984618101.

TECHNISCHE UNIVERSITÄT CHEMNITZ

Studentenservice – Zentrales Prüfungsamt
Selbstständigkeitserklärung

Name: **Shivam** Vorname: **Satyesh** geb. am: **08.08.1989** Matr.-Nr.: **386595**	**Bitte beachten:** 1. Bitte binden Sie dieses Blatt am Ende Ihrer Arbeit ein.

Selbstständigkeitserklärung*

Ich erkläre gegenüber der Technischen Universität Chemnitz, dass ich die vorliegende **Masterarbeit** selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keinem anderen Prüfer als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Datum: **14.11.2016** Unterschrift: Satyesh

---

* Statement of Authorship

I hereby certify to the Technische Universität Chemnitz that this thesis is all my own work and uses no external material other than that acknowledged in the text.

This work contains no plagiarism and all sentences or passages directly quoted from other people's work or including content derived from such work have been specifically credited to the authors and sources.

This paper has neither been submitted in the same or a similar form to any other examiner nor for the award of any other degree, nor has it previously been published.