

POLITECNICO DI TORINO

Department Of Control And Computer Engineering

Computer Engineering

Master Thesis

**Developing embedded automotive software
using the ISO26262 guidelines**



Supervisor:

Prof. Massimo Violante

Candidate:

Leonel Ngongang Tchamou

March 2018

Acknowledgement

First I would like to thank my supervisor and professor Massimo Violante, who throughout his several lectures and during this thesis work was always available to come with feedback, suggestions or interesting discussions, which served as inspiration for the direction of my career and this work.

I would also like to thank Mateo Serva from TXT e-Solution , who was always available to help me when I got confused .

I would like to thank the people at the TXT e-Solution , especially Concetta Argiri , who was always present and available to give very appreciated advice , encouragement and to solve any kind of administrative issue during my work.

Lastly I would like to thank my family, who even remaining far away was able to provide me with strength, confidence when I was feeling down or stressed. They helped keep my spirits high so that I could eventually finish with my study.

Abstract

This thesis work conducted in TXT e-Solutions spa company aims to design and implement an embedded configurable firmware Translation library so-called TRL that could be integrated into On-Board device (OBD) to gather vehicle diagnostic data collected from the CAN bus interface available on the OBD device port. This so-called TRL is implemented using the ISO 26262 standard guidelines in particular in order to guarantee the functional safety criticality and security of the software, a set of coding rules design by the Motor Industry Software Reliability Association (MISRA) have been adopted mainly the MISRA C guidelines. The design of this translation library implement the communication protocols needed to request diagnostics variable from the vehicle CAN bus and then translate these received raw data from vehicle dependent to vehicle independent information according to received rule from an external API server. Afterward the TRL design and implementation the testing of the software to validate that requirements functionality has been met, the testing is introduce based on the safety guideline ISO 26262 workflow that is from unit testing, module testing, to system testing.

Furthermore this thesis work introduce also a static verification analysis runs on the TRL software and some results metrics obtained in order to prove the absence of critical run-time errors under all possible control flows and data flows.

Key word : embedded configurable firmware - Translation library - diagnosis - CAN bus - OBD device -ISO 26262 - safety critical - MISRA C – Testing - static verification - run-time error .

Contents

<i>ACKNOWLEDGEMENT</i>	3
<i>ABSTRACT</i>	4
CHAPTER 1	8
INTRODUCTION	8
1.1. CHALLENGES AND PURPOSE	8
1.2. THESIS ORGANIZATION	9
CHAPTER 2	11
BACKGROUND	11
2.1. COMPANY PRESENTATION	11
TXT NEXT	11
TXT SENSE.....	11
2.2. EMBEDDED SOFTWARE TRENDS AND ISO26262	12
2.3. ON BOARD DEVICE STANDARD TECHNOLOGY	13
2.4. STATIC CODE ANALYSIS OVERVIEW	13
2.5. MISRA C RULES SUBSET OVERVIEW	14
2.6. POLYSPACE CODE PROVER OVERVIEW	15
<i>Value of Polyspace Code Prover Verification</i>	16
<i>How does Polyspace code prover works</i>	16
<i>What is Polyspace Tool used for</i>	16
CHAPTER 3	18
TECHNOLOGY AND TOOLS	18
3.1. SOFTWARE TOOLS	18
3.2. HARDWARE PLATE FORM.....	19
CHAPTER 4	21
TRANSLATION LIBRARY DESIGN ARCHITECTURE	21
4.1. SYSTEM ARCHITECTURE DESIGN OVERVIEW.....	21
4.2. TRL FUNCTIONALITY AND ARCHITECTURE	22
4.3. DIAGNOSTICS PARAMETERS DESCRIPTION AND READABLE METHODS	23
4.4. VDD FILE FORMAT DESCRIPTION.....	24

4.4.1. HEADER INFO	25
4.4.2. GLOBAL INFO.....	25
4.6. TRL LOGIC BLOCKS	26
4.6.1. TRL core management	26
4.6.2. TRL HLD DRIVER	29
4.7. SEQUENCE DIAGRAM OF INTERACTION BETWEEN DCF AND TRL	31
4.7.1. Initialization of the TRL.....	31
4.7.2. VIN reading operations	32
CHAPTER 5.....	33
TRANSLATION LIBRARY SOFTWARE INTERFACE AND DATA STRUCTURE.....	33
5.1. DATA STRUCTURES DESCRIPTION	33
5.1.1. Data structure for LISTEN ON CAN reading mode parameter	34
5.1.2. Data structure for REQUEST OBDII/EOBD protocol parameter.....	34
5.1.3. Data structure for REQUEST Do CAN (ISO 15765-4) protocol parameter.....	35
5.1.4. Data structure for REQUEST Volkswagen protocol parameter	35
5.1.5. Data structure for REQUEST OPEL protocol parameter	35
5.1.6. Data structure for REQUEST MINI protocol parameter.....	36
5.1.7. Data structure for OPERATION reading mode parameter.....	36
5.2. SOFTWARE INTERFACES	38
5.2.1. DCF-TRL Interface functions	40
5.2.2. Callbacks.....	41
5.2.3. VDD DECODER INTERFACE	41
5.2.4. VDD PARSER INTERFACE.....	42
5.2.5. Operation Manager interface	44
5.2.6. VIN READER interface.....	44
5.2.7. OBDII/EOBD interface.....	45
CHAPTER 6.....	46
EXPERIMENTAL TEST AND RESULTS	46
6.1. TEST ON PC BASED SIMULATED ENVIRONMENT.....	46
6.1.1. Test plan.....	46
6.1.2. Test Approach	46
6.1.3. Test description flow	47
6.1.4. Test Results.....	48
6.2. HARDWARE TEST	51
6.2.1. Hardware setup.....	51
6.2.2. Test description Approach.....	53

6.2.2.2. Setup of Can bus management tool for providing requested diagnostic parameter The Engine	55
CHAPTER 7.....	58
TRANSLATION LIBRARY SOFTWARE VERIFICATION WITH POLYSpace TOOL	58
7.1. TRL VERIFICATION WITH POLYSpace CODE PROVER AND MISRA C CHECK RULES.....	58
7.1.1. Project Setup.....	58
7.1.2. Project configuration : Coding Rule and Metrics.....	59
7.2. RESULTS ANALYSIS.....	59
CHAPTER 8.....	62
CONCLUSION AND FUTURE WORK.....	62
LIST OF FIGURES	63
LIST OF TABLES	64
BIBLIOGRAPHY	65

Chapter 1.

Introduction

This introductory chapter will present in general the issue this thesis aims to address , as well as the benefits that could be gained from the result of this work.

1.1. Challenges and Purpose

Modern automotive vehicles are highly complex systems, containing a large number of mechanical, electrical and electronic components. Recent development trends in the automotive industry integrates as many as 50-70 Electronic Control Units connected through several CAN communication networks for data monitoring , diagnosis and vehicle control.

The On-board Diagnosis (OBD) standard , defined as an interface with the external world to support vehicle maintenance with vehicle interior data , integrates devices to connect to engine and other subsystems through sensors and control actuators providing valuable source of information as vehicle monitor system. Information commonly available on the OBD port are retrieved based on OBD II standard and are on vary type either real time parameter such as Revolution Per Minute (RPM) , speed , pedal position , airflow rate , coolant temperature or Diagnostic trouble codes (DTC) or Vehicle Identification Number (VIN) or Number of miles driven with MIL on etc ... Concerned the OBD-II interface , five signalling protocols are allows based on which pins are present on the OBD connector. However most vehicles manufactures implement only one targeting a specific vehicle model.

This thesis aims to design and implement a new configurable embedded firmware library so-called Translation Library (TRL) to be integrated into OBD device and installed on the OBD port of vehicles that will gather vehicle diagnostic data collected from the CAN bus interface available on the OBD port. This TRL embedded firmware library is self-configuring regardless the vehicle model by understanding the vehicle model/version by means of the VIN number and implement the communication protocols needed to request diagnostics variable from the vehicle can bus and translate these received raw data from vehicle dependent to vehicle independent information according to received rule from an external API server . A specific format, named VDD (Vehicle Diagnostics Description file), should be responsible for programming the firmware library in order to collect relevant vehicle data through the can bus. This VDD file format has the purpose of describing the methods needed for reading predefined diagnostic parameters in a specific vehicle model/version.

However the development of such related automotive embedded software to be integrated into the vehicle network compose of several of Electronic control units controlling safety critical functionality of the vehicle such as the braking system where consequences of failure could result in the loss of human lives, require to adopt some stringent standards that guarantee the functional safety criticality and security of the software. Actually the verification and validation of these software is performed using extensive testing and simulation based which is a process that is both time-consuming and error-prone.

In order to address this issue, the International Organization for Standardization (ISO) released a functional safety standard titled Road vehicles Functional safety in November, 2011. This standard known as ISO 26262 and establishes guideline and requirements on the development process of automotive software systems in order to guarantee the functional safety of such system. In particular one of the guideline required by the standard is the software compliant with a set of rules produced by the Motor Industry Software Reliability Association (MISRA) which aim to increase the safety, portability and reliability of code artefacts written in the C language. However recently the development of development of safety-critical electric/electronic automotive systems is performed by an increasing number of software tools very high sophisticated , for instance Polyspace tool , in order to meet these coding guideline rules requirement compliant with ISO 26262 safety standard.

Therefore this thesis also introduce a static verification analysis run on the TRL software and some results metrics obtained in order to prove with some degree it safeness and the absence of critical run-time errors under all possible control flows and data flows.

1.2. Thesis organization

This thesis work is organised in mainly two parts . The first part of the work will concert the design and implementation of the Translation library (TRL) according to the traditional workflow using in software development process then the software will be tested either in simulation on development tool or on the physical hardware to verifying the actual behaviour of the system. While the second part will attempt at verifying the software quality by running the static analysis verification for “0” defect run time error and for compliance with one of the guideline require by the ISO 26262 standard namely MISRA C 2012 coding rules guideline.

More in detail following this introduction part , chapter 2 will first give a brief presentation of the company where my work has been conducted then will follow a brief summary background of

trend in embedded software and a summary of the ISO 26262 standard. Afterward a brief background of OBD technology will be presented . Static analysis tool commonly used for software verification will be introduced , then MISRA C coding guideline will be briefly summarized and finally a brief summary on Polyspace code prover for static code analysis will be presented.

Chapter 3 will present technologies and tools used in the development process and testing either software tool or hardware tool.

Chapter 4 will summarize the complete TRL design architecture . In particular this chapter will present the vary building block composing the TRL system . However the chapter will more enter in detailed to describe the internal description of block related to my own implementation.

Chapter 5 will present the software architecture of the TRL , the related interfaces and the logic that allow the software to run . In particular it will present the description of data structure and software interface. However this chapter will also give a deep detail just to the part related to my own implementation.

Chapter 6 will describe the vary test activity performed in other to validate the main requirement functionality of the TRL. In particular it will describe focus on two kind of test namely the software test on the development tool and the hardware test on the real hardware setup and the vary result will be presented.

Chapter 7 will introduce the embedded software verification with static analysis tool namely Polyspace software tool. And finally will be present some results obtained by verification of the TRL software.

Chapter 8 will present the final conclusion and will highlight some future work.

Chapter 2.

Background

2.1. Company presentation

Founded in 1989 , TXT e-solution is a leading international software products and solution vendor , leader in “ Strategy Enterprise Solution” with consolidated revenues around € 69.2 millions since 2016 .

TXT is based in Milan and has branches in Italy, Berlin (D), London (UK), Paris (France), Seattle (USA) and Chiasso (CH). TXT is specialized in the most dynamic and agile markets with the highest degree of innovation and renewal that require state-of-the art solutions, where process and business play a fundamental role in growth and competitiveness.

To answer to customer's needs TXT has always invested in technology and in business processes innovation, making this a cornerstone on which to build its success

TXT Group serve key high-technology markets through its business divisions TXT NEXT and TXT SENSE

TXT NEXT

Provides Advanced software engineering services for companies in the Aerospace , Automotive , High-Tech manufacturing and Banking sectors to empower their engineering capabilities.

TXT NEXT support customers throughout their product lifecycle. From software design and development to verification, integration and final certification, offering engineering services in the field of :

On-Board Software

Simulation & Modelling

Complex Manufacturing

Business Intelligence & Business Process Management

Mission Critical Systems software development

Independent Verification and Validation

IT Governance & Quality

TXT SENSE

TXT Sense, a new Division of TXT e-solutions, it develop and market innovative applications of New Augmented Reality to many service and industrial sectors.

The application sectors which it's addressed by TXT Sense include luxury and fashion, advanced manufacturing, medical, retail, media and advertising .

2.2. Embedded software trends and ISO26262

Embedded software can be seen as a software system designed tailoring specific application running on a custom hardware. However very often when we speak about software , we mostly think to IT system , such as general-purpose PCs , online Internet applications services . However such system embeds just less than 2 percent of the produced microprocessors.

Nowadays most embedded software processors are produced most for system such as mobile phone, washing machine, pacemaker , aircraft , robots , cameras. This trends toward embedded software is accelerating and the global market for embedded systems is more than 160 billion euros, involving approximately 3 billion embedded units delivered per year and a compound annual growth more than 10 percent. Embedded software is anticipated to attain \$18.61 billion by 2023 driven by the steady growth in production of consumer electronic devices and increase investments in automation technologies in the manufacturing sector.

Nevertheless embedded software development poses an extraordinary constraints such as real time; the embedded software system's timing must provide the expected action within a maximum specified time under all circumstances, they pose also the problem of Reliability in such a way that the system have to operate for long time without unexpected behavior. The Safety related should also be guaranty, the security assurance must be also provided to avoid life threatening situations , and finally very limited among of resource such as small memory space will be well manage. Current trends in embedded systems also focus more on how to manage the increasing software content, with a strong emphasis on standardization of the embedded software structure. The rapidly increasing complexity of embedded software development is one of the most important challenges for increasing product quality, reducing time to market, and reducing development cost. Model Driven Design is then one of the promising approaches that have emerged over the last decade in which developers instead of directly coding the software, they model software systems using intuitive, more expressive, graphical notations, which provide a higher level of abstraction than native programming languages. In this way, generators automatically create the code implementing the system functionalities . Therefore nowadays most embedded software development processes are shifting to model-based development

concept based on the ISO 26262 safety standard that reflect more the importance of this approach in automotive software development.

2.3. On Board Device standard technology

The idea of OBD was first proposed in California in 1984 to detect engine operation conditions for air-pollution monitoring, and then, it became a subsystem into American vehicle in 1988. The first standard, which is known as OBD I , contains Basic proprietary instrument such as signal light, storage, and indication of diagnostic trouble code (DTC). However, different manufactures designed and implemented different interface sockets, codes, and functions that brought up confusion to the maintenance technicians.

In 1988, OBD II was proposed in California by the Society of Automotive Engineers (SAE) and International Standardization Organization (ISO) which described the interchange of digital information between ECUs and a diagnostic scan tool.

A basic OBD system consist of an ECU (Electronic Control Unit), which gets input from sensors (e.g. carbon sensors) to control actuators (e.g. fuel injectors) to get the desired performance. Modern vehicle can support hundreds of parameters, which can be accessed via the **DLC** (Diagnostic Link Connector) using a device called a scan tool.

Various tools are available on the market that allows plug in OBD connector to access OBD range functions like hand-held scan tools, pc-based scan tools.

2.4. Static code analysis overview

Static Analysis is a way of examining program code and reason over all possible behavior that may arise at run-time without carry out the source code . It's usually performed as part of a Code Review and could be carried out at the Implementation phase of a safety Security Development Lifecycle.

Technique behind the static code analysis are well known such as Data Flow Analysis used to collect run-time information about data in software while it is in a static state.

Tools based on static analysis can be used to find out defects in the source.

Static analysis tools compare to manual reviews , which are performed by human auditors using methods such as self-review of the written code ; walkthrough focusing on the presentation to an audience of the

code in question by its programmer ; peer review that is when he programmer presents his code to a colleague to review and finally inspection and audit which is usually done by a third party of evaluators , static analysis are faster which means that source code can be evaluate more frequently and could built-in some knowledge required to perform this type of source code analysis and have the same level of expertise as a human auditor .

Although more advanced tools for static analysis are nowadays introduced , static analysis cannot solve all of the safety and security related problems, mainly because these tools look for a fixed set of patterns, or rules in the code , that means the output of the tool still require human evaluation. So the tool can sometime produce false negatives (the program contains bugs that the tool doesn't report) or false positives (the tool reports bugs that the program doesn't contain). One of the most advantage using static code analysis compare to dynamic analysis is that result of the code analysis produced are generalized for future execution steps. Tools based on static analysis commonly used for can be classified in the following categories: Microsoft .NET, Java, C/C++ and Multi-Language. In addition, some tools are either open-source or commercial ones. In the following there are list of some tools for static code analysis classified by category.

- Microsoft .NET : FxCop , **CodeIt.Right**
- JAVA : FindBugs , PMD , CheckStyle , Jlint
- C/C++ : PRQA , Polyspace , Visual studio , Sparse etc..
- **Multi-Language** : Coverity Prevent , RATS , Understand .. .

2.5. MISRA C rules subset overview

MISRA which stand for Motor Industry Research Association is an international consortium of major car and car component manufacturers originally based in UK . In 1998 this association, due to the proliferation of consumer embedded control system in particular in safety critical application where failure can lead to human or environment damage for instance high-tech medical scanner, produced an official set of guidelines for the use of C language in automotive electronic systems [8] with a goal of eliminating as many fault modes as possible. The guidelines describe a restricted subset of C language defined by a number of rules backed by informal explanatory text and some code fragments. Then these guidelines have become widely used around the world , not only in the automotive industry but also in industry which there is a safety and security-critical component , such as aircraft and medical devices.

MISRA C standard coding guidelines is made of 93 required rules together with 34 advisory or recommendations rules.

Although this safer subset of rules has been successfully appreciated worldwide yet it suffer of some drawbacks such as rule incorrectness or rule redundancy therefore some time they to be justified and prove the conformance to the subset.

MISRA C coding guidelines subsets has been produced mainly to promote a common coding style among c programmers in such a way that they would find it much easier to read their colleague's code and that this would contribute to an improvement in response time for changes, either corrective, (fixing defects), perfective, (cleaning up without changing functionality), or adaptive, (adding new features) in this way most commonly mode failure can be prevented.

2.6. Polyspace Code Prover overview

Polyspace product is a software tool that allow verification of C,C++ , Ada source code by detecting run-time error without the software source code is compiled and executed.

Polyspace Code Prover is a solid static analysis tool that proves the absence of overflow, division-by-zero, out-of-bounds array access, and certain other run-time errors in C and C++ source code.

In order to verify source , the verification parameter have to be set up , then verification have to be run and thanks to its integrated graphical user interface results can be efficiently reviewed .

To prove that no run-error occurs during the process of verification , Polyspace Code Prover tool Applied a semantic analysis and an abstract interpretation of the source based on formal methods to verify software control and dataflow . It also based on semantic color coding analysis by assigning color to operations in the source code.

Polyspace Code Prover can speed up the verification phase by displaying range information for variables and function return values, and can prove which variables exceed specified range limits.

Results can be published to a dashboard to track quality metrics and ensure conformance with software quality requirement.

The color-coding helps to quickly identify run-time errors and find the exact location of that error in the source code. After the errors is identity and fixed , the verification can be easily run again.

The different coding colours used by the tool are specify as following

- **Green** : Indicates that the operation is proven and reliable to not have certain kinds of run time- error.
- **Red** : Indicates that the operation is proven to have at least one run-error.
- **Grey** : Indicate dead code that is unreachable code
- **Orange** : Indicates that the operation is unproven and can have run time error along some execution paths depending on the input of an operation phase.

Value of Polyspace Code Prover Verification

By using Polyspace verification software for design verification , it can significantly help to the reliability of the application software by prove code correctness and identifying run-time errors. Moreover The development time could also being significantly reduced due to the automated verification process and efficiency review verification result : color coding , distribution graph . with Polyspace we can know the parts of the code that do not have errors, and we can only focus on the code with proven (red , grey code) or unproven (orange code) .

How does Polyspace code prover works

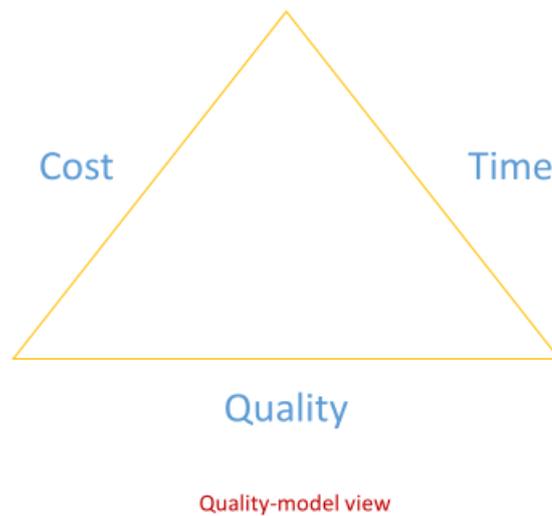
Polyspace code prover software uses static code verification (see ...) to prove the absence of run-time errors. Moreover it uses formal method based on abstract interpretation that is take into account all possible input value into an function and test or verify it with all possible operation and combination in the source code . This mean that the data flow of a variable is taken into account to prove a property and that's the power of this method compare to other statics analysis tool

What is Polyspace Tool used for

- **Early Defect Detection for Software Quality and Productivity**

Most software development teams have as main goal to maximize both quality and productivity of the software. However, when developing software, there are three related variables to consider: cost, quality, and time. Changing the requirements for one of these variables affects the other two. Here the issue is that the criticality of the software application determines the balance between these three variables: quality model . However, with classical testing processes, development teams generally try to achieve their quality model by testing all modules in an application until each module meets the required quality level. Unfortunately, this process often ends before quality requirements are met, because the available time or budget has been exhausted.

With Polyspace the quality and productivity of the software can be improve at the same time by integrating verification into the development process, considering time and cost restrictions.



- **Coding Standards**

Polyspace software allows to analyze the source code in order to demonstrate compliance with established C and C++ coding standards. In particular the tool can guarantee compliance with coding standards as MISRA C 2004, MISRA C 2012 and MISRA C++ guidelines.

The advantage of applying coding rule guideline is that it can reduce significantly the number of defect and so improve the quality of the software. When MISRA C rules are violated, Polyspace software provide messages with information about the rule violations. Most messages are reported during the compile phase of an analysis

- **Certification**

- ISO 26262
- DO-178B
- IEC 61508

Chapter 3.

TECHNOLOGY and Tools

In This chapter we will briefly introduce the different tools and technologies that have helped to accomplish this thesis work.

The word Technology can means any kind of knowledge of technique or processes , either standalone or embedded into a machine of a tool , to allow operation without detailed knowledge of their working. In the following the necessary tools used to achieve the goal of this work are classified in category of software tools , Hardware plate form and collaborative tools.

3.1. Software tools

- Team Foundation server (TFS) and Visual studio Professional 2017 By Microsoft
Visual studio in particular have been used for the software development phase of this work and mainly used for PC based simulation of the software application.
The TFS tool acts as collaborative tool to support the team work that have contributed to accomplish the goal of this work.

- Eclipse for DS-5 By ARM
It's an *Integrated Development Environment* (IDE) that combines the Eclipse IDE from the Eclipse Foundation with the compilation and debug technology of the ARM tools. It provide project manager , Editor for C/C++ or ARM assembly language.
This tool have been to compile the C code of the implemented application to be run on the target hardware.

- Polyspace tool by Mathworks
Static code analysis tool for large-scale analysis by abstract interpretation to detect, or prove the absence of, certain run-time errors in source code for the C, C++, and Ada programming languages. The tool also checks source code for adherence to appropriate code standards.
This tool have been mainly used to check the implemented application source compliant with Misra C 2012 rules.

➤ Canalyzer software Tool by Vector

It's an universal software tool for ECU network and distributed systems. It makes easy to observe and analyse data traffic in Physical layer such as CAN , LIN , MOST or Flex Ray systems. The tool optimally covers all application areas from a simple network analysis to a high-performance analysis and emulation system.

CANalyzer can be used in many phases of development and use of diagnostics in control units.

- Diagnosi of ECUs
- Specification/integration/regression test
- Analysis of the communication of real control units
- Troubleshooting

The following picture a view the software tool

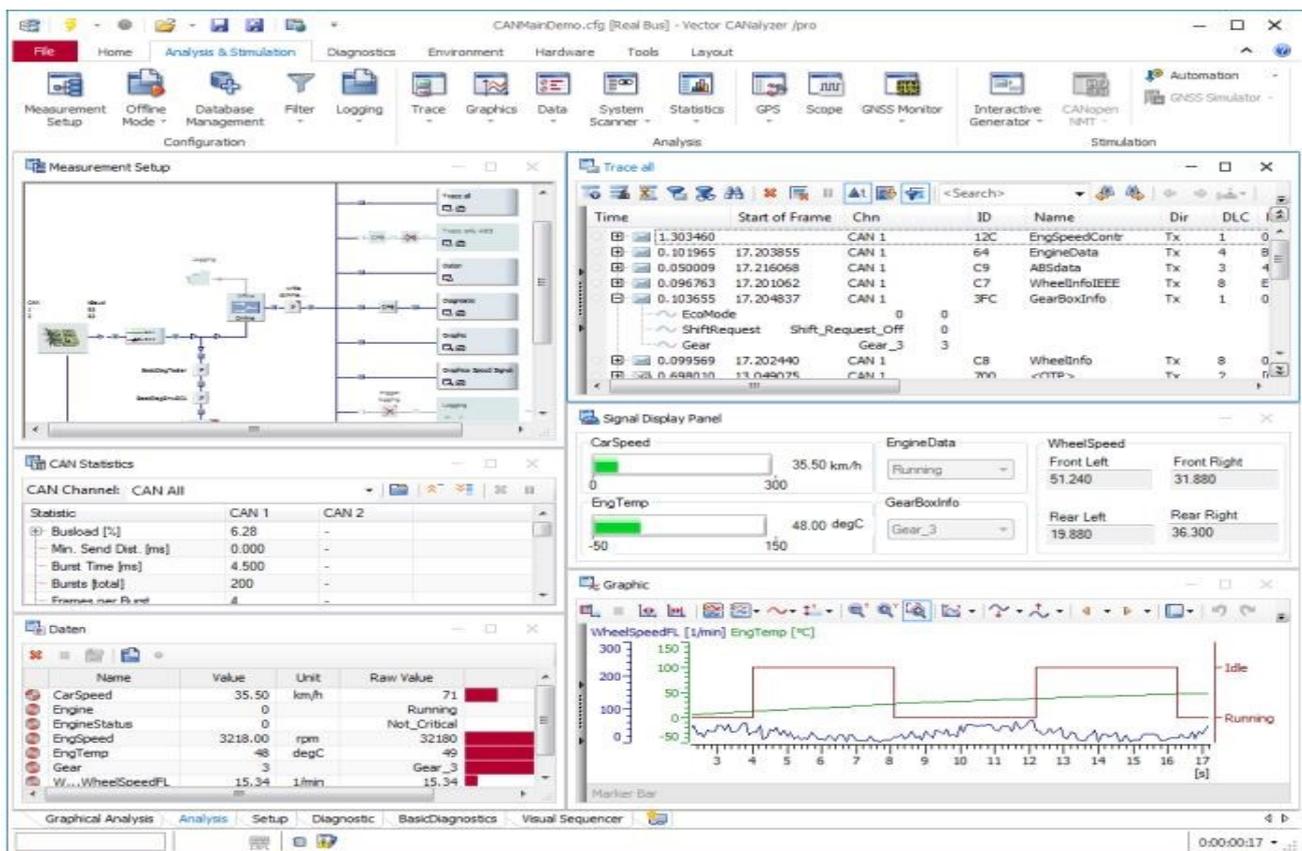


Figure 1: CanAnalyzer tool

3.2. Hardware Plate form

➤ Teseo III GNSS evaluation board

Teseo EVB board is a complete standalone evaluation platform for Teseo III GNSS ST solution.

Teseo III embeds the high performance ARM946 microprocessor with dedicated SRAM and several serial communication interfaces, including USB, SPI, I²C, UART and CAN.

Performance and configuration can be analysed using the ST TESEO-SUITE PC Tool.



Figure 2 :T3 EVB

EVB-T3

➤ *CANcaseXL*

The CANcaseXL is a USB interface with two CAN controllers which can send and receive CAN messages with 11 bit and 29 bit identifiers as well as remote frames without restrictions.

Additional, the CANcaseXL is able to detect and generate error frames on the bus. Various transceivers are available to interface the CANcaseXL to a particular type of bus, one of CAN and LIN transceivers is Piggybacks transceiver .



Figure 3: CanCaseXL

Chapter 4.

Translation Library Design Architecture

This chapter will briefly summary the system architecture design of the complete application and its related interfaces, then will more zoom inside our interested design namely the Translation Library (TRL) and describe it architecture logic block and internal interfaces. Moreover, the sequence diagram illustrating the interaction between the external Data collection layer and TRL will be presented and finally the sequence diagram of internal operation of the TRL about VIN reading algorithm will be describe.

4.1. System Architecture design overview

The top level view of the system architecture design as show in Figure 4, describes the overview design application containing the TRL. Actually the system is composed by :

1. The Central Server that instruct the On-board Device on how the reading vehicle parameters will be done on specific vehicles through a configuration file. The interface for communication between central server and on-board Device is through GPRS o mobile network API which the description are out of the scope of this work.
2. The On-board Device (OBD) which is responsible of gathering diagnostic parameters from the CAN bus using the rules received from the central server and it is also responsible of sending data back to the server for further processing. The OBD device from one side

interface the central server through GPRS API and from another side interface the physical can bus through the CAN driver API from ARM .

3. The Can Bus On OBD port that when connected to the vehicle Can network provides the requested diagnostic parameters from that specific vehicle.

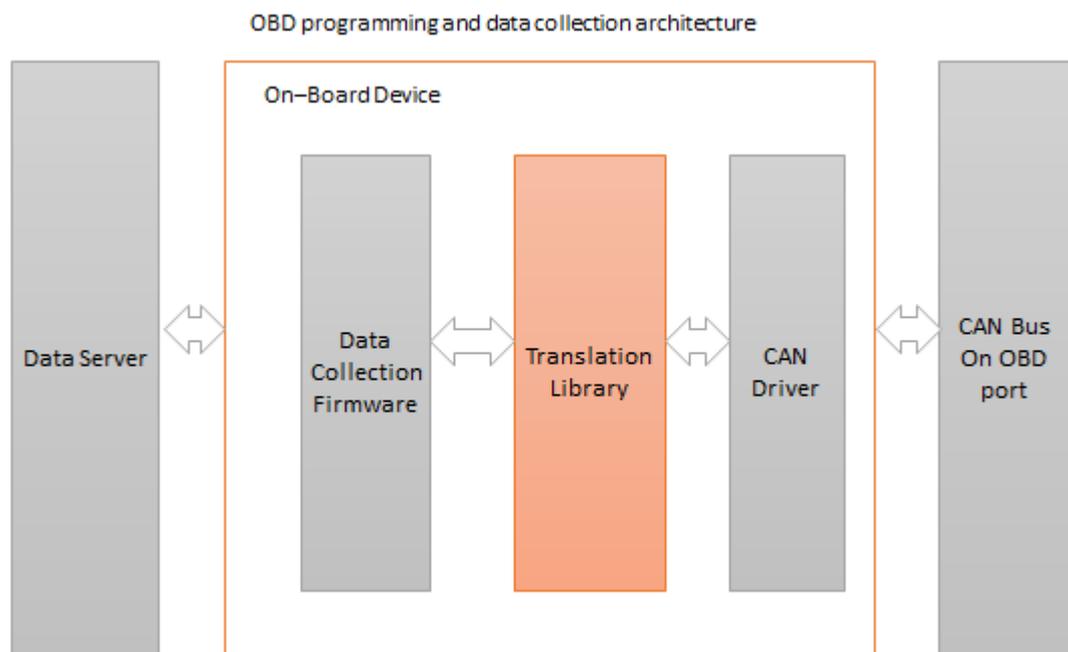


Figure 4 : OBU Design architecture

4.2. TRL functionality and architecture

The Translation Library firmware that serves as an interface between the can bus and the Data Collection Firmware embedded into the OBD device has the main functionalities of implementing the protocol needed for request diagnostics parameters from the vehicle can bus, and then to translate those vehicle-dependent data into vehicle-independent information and pass such translated information to the data collection firmware through an Application Programming Interface (API) . The Data Collection Firmware, which the main role is to communicate with the central server, is responsible of sending these vehicle-independent information to the central server relying on GPRS or mobile network API for further processing or presentation.

The system is required to be self-configuring by understanding the vehicle model/version by means of the VIN number ,described later, that the OBD device will be send by to the Central server which will provide back the right configuration file namely the Vehicle diagnostic

description (VDD) file to the OBD device which will store it to its Non Volatile Memory and use it, when an actual parameter reading should be done.

The specification considers diagnostic parameter readable over CAN bus through international standardized or manufacturer-specific protocols. The following Table 1 : Normative Standards are applies :

Normative	Description
ISO 11898-1	DataLink layer of Controller Area Network (CAN)
ISO 11898-2	Physical Layer of high-speed CAN
ISO 15765-2	Diagnostic over Controller Area Network (Do CAN) - Transport protocol and network layer services
ISO 15765-4	Diagnostic over Controller Area Network (Do CAN) - Requirements for emissions-related systems
ISO 15031-5 / SAE J1979	OBDII / EOBD - communication between the vehicle's OBD systems and test equipment within the scope of the legislated emissions-related OBD
ISO 14230-3	Diagnostic systems - Keyword Protocol 2000 - Part 3: Application layer
SAE J2819	Diagnostic communication protocol TP2.0

Table 1 : Normative Standards

4.3. Diagnostics Parameters Description and Readable Methods

Each diagnostic parameter has an ID that identify uniquely the parameter in order associate to it the proper parsing/decoding rule from CAN bus or which allow the central server to request its collection to the OBU. This ID of the diagnostic parameter is 16 bits fixed where the first 4 most significant bits identifies the Reading methods to be used for the reading over the CAN bus of the parameter and the last 12 remaining bits identify the name of the parameter , see the following Figure 5: parameter ID



Figure 5: parameter ID

In the following there is a List of some possible diagnostic parameters name.

1. Lubricant Temperature level
2. Engine RPM
3. External Temperature
4. Selected Gear
5. Distance travelled with MIL ON
6. Battery level
7. ...

Each possible diagnostic parameters is characterized by its reading methods classified as belonging to one of the following type :

- Listen On Can : Diagnostic Parameter readable, without enquiring, on the CAN network.
- REQUEST ON CAN : Diagnostic Parameter readable through a diagnostic protocol using the CAN bus as physical and data link layer.
- OPERATION_: Parameters generated by a calculation on one or more inputs, that have reading modes LISTEN or REQUEST or another OPERATION

4.4. VDD File Format description

The vehicle diagnostic description file is a binary file containing information that are strictly necessary to read and decode a diagnostics parameters. It's compose of an header section , a global info section and the list of LISTEN , REQUEST , OPERATION for a parameters .

This file description represents the configuration file of the TRL firmware layer.

The following Figure 6 : VDD FORMAT. shows how the format look like



Figure 6 : VDD FORMAT.

4.4.1. HEADER INFO

The header info is represented by the first 11 bytes of the VDD file . It contains information about the file content and the format version coded through an ASCII string. It's shape as following : VDDXXXYY.YY were XXX are three characters that indicate the file content and the possible value is PAR stating that the file contains the description of LISTEN, REQUEST, OPERATION parameters reading methods. The YY.YY are five characters that indicate format version for instance 01.00

4.4.2. GLOBAL INFO

After the header , follows the global info , that state the number of vehicle channel data buses that an OBU could access and the ID of the channel and finally the channel speed.

4.5. VIN Format description

The VIN Figure 7 is a 17 characters alphanumeric hierarchical code that uniquely identify the vehicle as specify in the ISO-3779 and ISO-3780 standard. VIN contains 3 sections: WMI, VDS, and VIS.

The WMI is 3 digits long and uniquely designates the manufacturer's continent, country, and the unique national identify.

The VDS is 6 characters long and describes the vehicle attributes (weight, model).

The VIS is 8 digits long. Combined with WMI, they uniquely identify a vehicle worldwide.

VIS ranges from the 10th digit to the 17th. Digits (10-13) are alphanumeric and (14-17) numeric.

In order to assist in the identification of vehicles and to help prevent vehicle theft, VINs are typically affixed to different parts of the vehicle.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
World manufacturer identifier			VDS						VIS							

Figure 7: VIN

4.6. TRL logic blocks

The main intended functionalities of the Translation Library firmware are resumed in its architecture design diagram in . It's based on logical blocks diagrams each implemented a specific function or protocol.

The TRL architecture is decomposed into two major blocks part .

- TRL core management
- TRL High Level Driver (HLD)

The TRL core management and the TRL HLD are interfaced by means of the TRL HLD interface

4.6.1. TRL core management

The Translation Library core management from one side by means of the TRL- HLD interface is responsible of gathering vehicle dependent data coming from the TRL High Level and translate it to vehicle independent data to be sent to the central server through relying on the data collection interface. And from another side by means of DCF-TRL interface it receives a configuration file and is responsible of setting the logic how the diagnostic parameter should be read. The summarized architecture is described as following Figure 8.

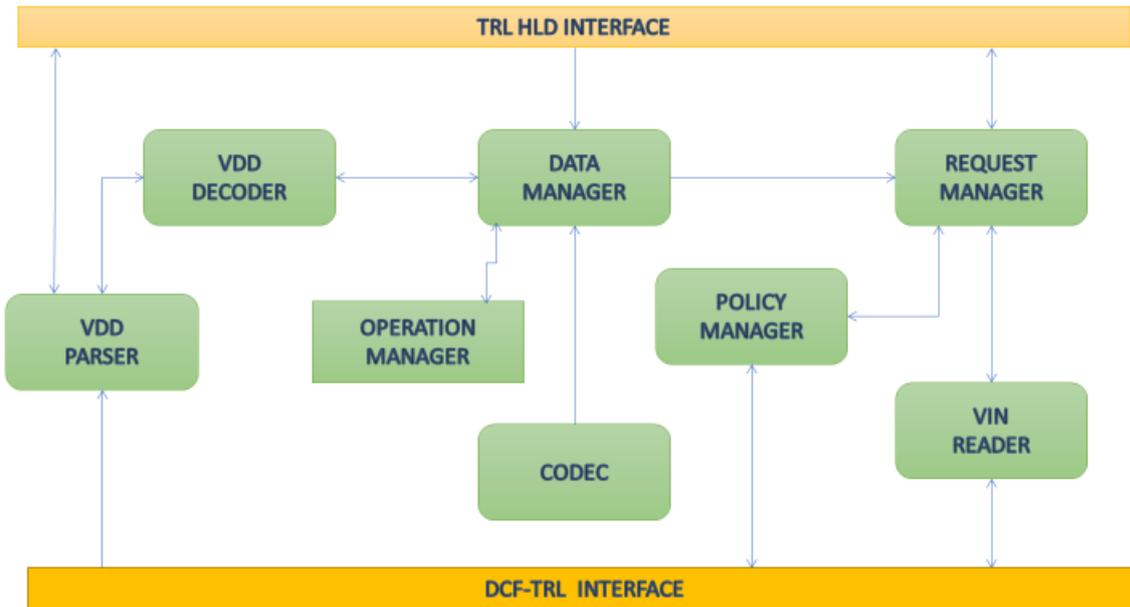


Figure 8: TRL-Core

The TRL core management is further decomposed in the following parts :

DATA MANAGER

This block is a container for the data to be processed and for the rules used by the Translation Library to retrieve information from the CAN bus.

VIN READER

This block implements the algorithms needed to read the VIN number Figure 7 by using hardcoded data or the Vehicle Diagnostic Description file received from the Data Collection Firmware.

POLICY MANAGER

This is responsible for executing the policy rules that have been configured.

VDD PARSER

This block parses the Vehicle Diagnostic Description raw data file received from the server and extracts its data into the DATA MANAGER to be used by the other blocks

VDD DECODER

This block implement the logic that allow from VDD file first to peek the protocol type to be used for requesting the diagnostic parameter then based on that protocol type , trigger the parsing for loading the corresponding data structure and the finally prepare the parameter ready to be use by the data manager , by configuring the data structure in the data manager

OPERATION MANAGER

This block is responsible for evaluating diagnostics parameter of type Operation in reverse polish notation and then responsible of performing logic or mathematical operation as enumerated in the following list.

1. Operation NOP
2. Operation Calculate MIN
3. Operation Calculate MAX
4. Operation Calculate AVERAGE
5. Operation Calculate RATE
6. Operation Calculate TIME
7. Operation Calculate DISTANCE
8. Operation Compare with a RANGE
9. Operation GREATER THAN A VALUE
10. Operation LOWER THAN A VALUE
11. Operation Bitwise NOT
12. Operation Bitwise AND
13. Operation Bitwise OR
14. Operation SUM
15. Operation DIFFERENCE
16. Operation MULTIPLICATION
17. Operation DIVISION
18. Operation DEBOUNCE

CODEC

This block implement the logic needed to convert RAW data into engineering variable. It job is first extract the relevant portion of the message that is received from High Level Driver and reorder the message byte according to the endianness. Then Apply a linear transformation on a value and finally remap these value according to the provided map

REQUEST MANAGER

This block is responsible of handling any request from user and forward the user request to the High Level Driver through the data manager for retrieving or sending message on the can bus.

4.6.2. TRL HLD driver

The HLD (High Level Driver) implements the diagnostic protocols needed to communicate over the CAN Bus and then allow the reading of diagnostics parameters regardless of the vehicle model or version. The reading of these requested diagnostics parameters is based on a specific standard and the available are highlighted in the following Table 2.

The architecture design of the HLD is shown in the higher part of the TRL VIEW

Diagnostic Protocol	Description
OBDII/EOBD	request/response exchange, as stated in OBDII and EOBD standards
DoCAN	Diagnostics on CAN standard (ISO 15765-4) and little variations of it
VW	Volkswagen specific diagnostic protocol (TP2.0 standard)
OPEL	Opel specific diagnostic protocol
MINI	Mini specific diagnostic protocol

Table 2: Diagnostic Protocol

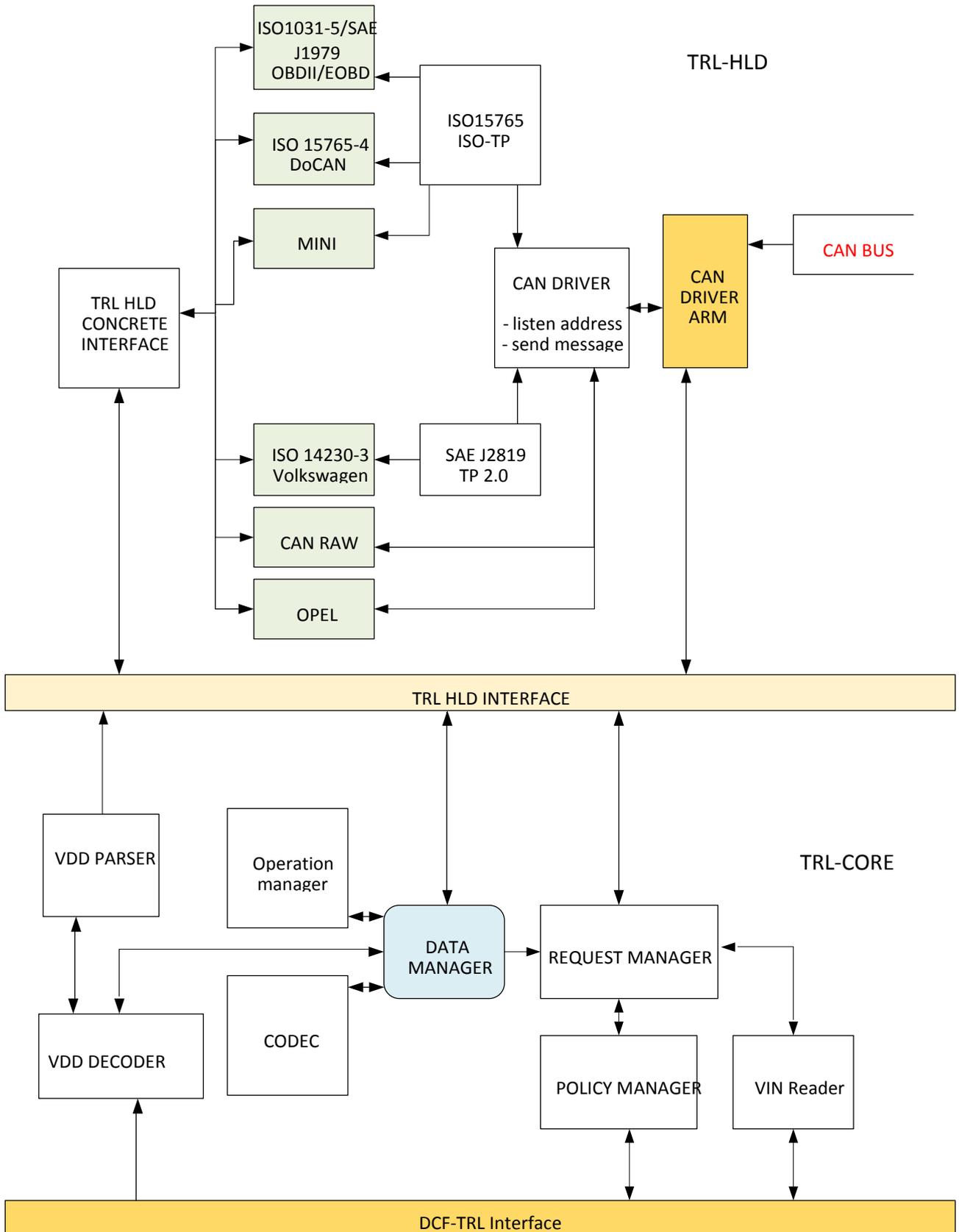


Figure 9 : TRL VIEW

4.7. Sequence diagram of Interaction between DCF and TRL

4.7.1. Initialization of the TRL

The following sequence diagrams shows the interaction between the DCF layer (on the left) and the TRL library (on the right) see DCF-TRL sequence diagram.

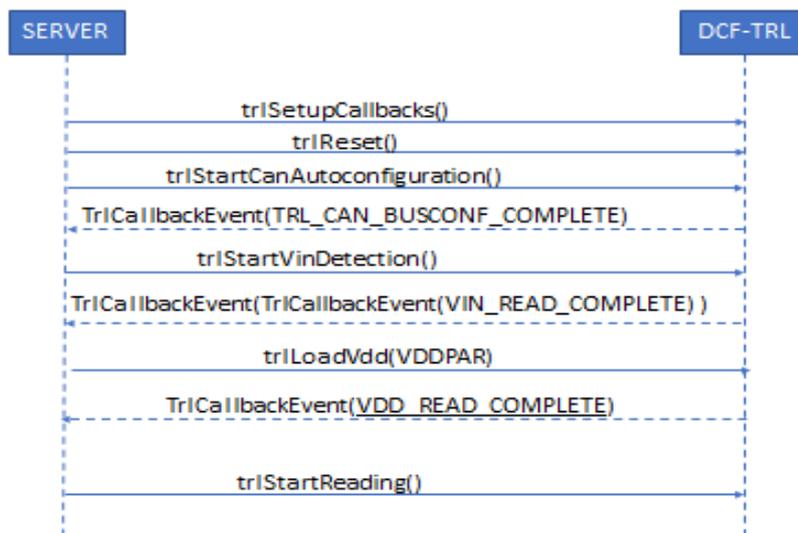


Figure 10:DCF-TRL sequence diagram

At the start up , after call-back have setup and the TRL has been reset , the DCF can ask to auto-check the CAN bus rate. After auto-detection of the CAN rate the DCF save in flash the proper values. If the DCF retrieves the CAN rate value at start up, it passed to the TRL in order to skip the auto-detection.

In case the DCF has no previously detected VIN information stored in flash, it asks to TRL to detect the VIN (trIStartVinDetection). In case of positive detection (call-back VIN_READ_COMPLETE) the resulting VIN is stored in flash by the DCF.

In case of correct VIN detection, the VDDPAR is provided by the DCF to the TRL (trILoadVdd) (DCF retrieves this information from previously saved in flash VDD or asking it to the server). The TRL analyses the VDDPAR and provides the indication of correct read or not (call-back

VDD_READ_COMPLETE). The DCF requires the starting/stopping of the acquisition session of the diagnostics parameters (trIStartReading/trIStopReading).

4.7.2. VIN reading operations

The following diagram details the TRL internal logic about VIN detection algorithm.

After correct detection of the VIN, the DCF save the VIN in the flash memory.

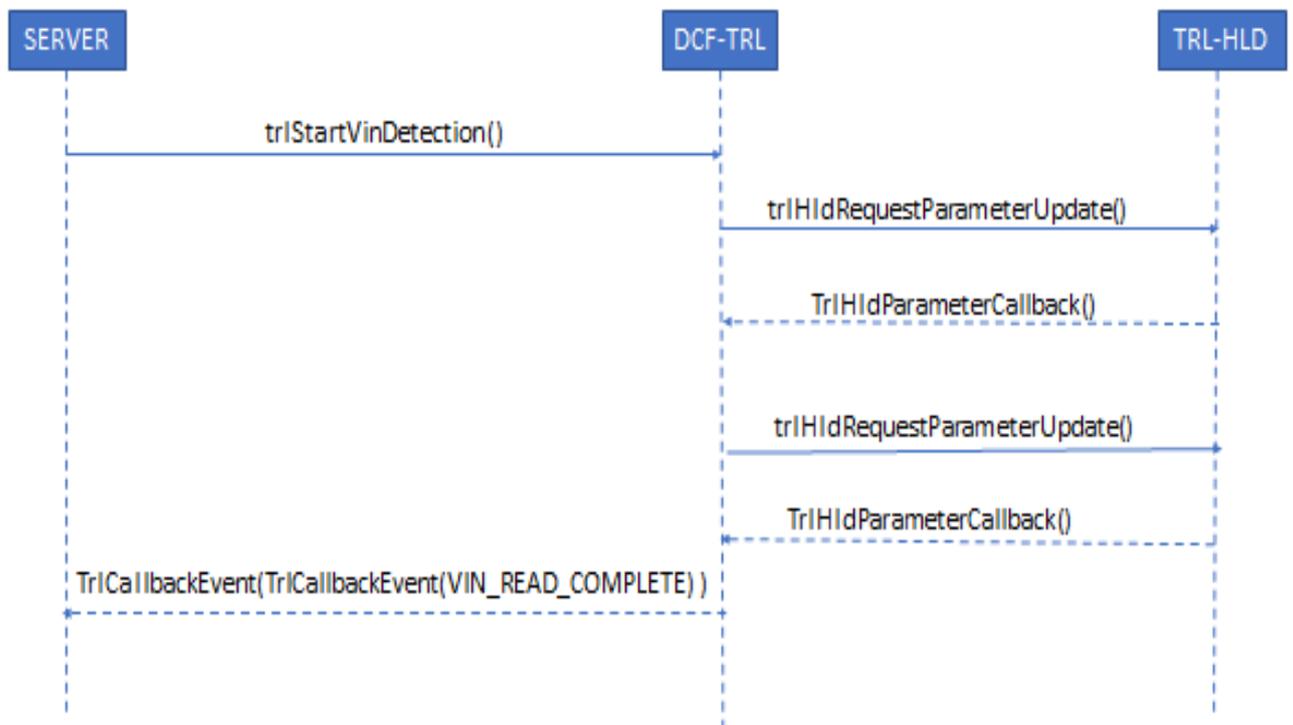


Figure 11:VIN Detection sequence diagram

Chapter 5.

Translation Library Software Interface and Data Structure

This chapter will describe more in detail the software implementation of the TRL according to the architecture specified in the previous chapter.

In particular this chapter will deeply focus on the implementation blocks I have designed and presented in the architecture of the TRL. Moreover additional interface to other block of the TRL architecture will be briefly presented.

5.1. Data structures description

This section will describe the data structure defined to exchange information between DCF and TRL and between TRL and vehicle CAN bus allowing the TRL to read and decode each diagnostic parameters.

The data structure are defined based on specific diagnostic protocol associated with it proper reading method which could be

- LISTEN : That is reading method of vehicle diagnostic parameters, based on parsing basic protocol packets (mainly CAN) that are exchanged on the vehicle data networks between the ECUs.
- REQUEST : That is reading method of vehicle diagnostic parameters, based on a diagnostic protocol that is made up of one or more requests by External Test Equipment and one or more answers by a vehicle ECU.
- OPERATION : That is reading method of vehicle diagnostic parameters, based on a calculation that uses one or more inputs, that could be other diagnostic parameters.

Moreover all data structure that allow the TRL to read and decode diagnostics parameters have common fields that are describe as following

```

struct VddDecodeParameterConfig
{
    u16 PAR_ID; Diagnostic Parameter Identification number
    u8 DPRG_SOURCE; Parsing Rule Information source
    u8 CHANNEL_ID; vehicle data bus channel
    u16 STARTBIT; position inside the CAN message payload of the requested
parameter raw value
    u16 LENGTH; length, in number of bits, of the requested parameter raw
value inside the CAN message payload, counted starting from STARTBIT

    enum Vdd1RawValueType RAW_VALUE_TYPE; type(Boolean, Signed, ASCII string
etc)of the raw value of the parameter as present in the CAN message

    enum Vdd1FinalValueType FINAL_VALUE_TYPE; type and length of the final
format of the parameter value that the TRL should give back to the DCF

    enum Endianess ENDIANESS; Byte order
    u8 FACTOR_PRESENT; Factor needed to convert raw value
    u8 OFFSET_PRESENT; Offset needed to convert raw value
    u8 SPARE;
    float FACTOR;
    float OFFSET;
    struct ValueEncoding VALUE_ENCODING; Encoding rules for value encoded
parameters
};

```

5.1.1. Data structure for LISTEN ON CAN reading mode parameter

The information for describing how to read and decode a LISTEN ON CAN parameter type are listed and comments in the following couple with the above described field

```

struct ListenOnCanParameters
{
    enum Can_ID_format CAN_ID_FORMAT; format of the ID of the CAN message to
capture(standard or extended)
    u32 CAN_ID; ID of the CAN message to capture
    u8 RECOGN_SEQ_LEN; CAN Message Recognition Sequence Length
    u8 RECOGN_SEQ_STARTBIT; CAN Message Recognition Sequence Position as
start of bit string
    u8 RECOGN_SEQ_DATA[8]; CAN Message Recognition Sequence Data
};

```

5.1.2. Data structure for REQUEST OBDII/EOBD protocol parameter

```

/* Data structure for OBDII/EOBD REQUEST parameters*/
struct RequestOBD_EOBDParameter
{
    u8 ID_PROTOCOL;
    enum Can_ID_format CAN_ID_FORMAT;
    u8 REQ_DATA_LEN; /* the data length should be express in numer of byte*/
    u8 padding;
    u8 REQ_DATA[8]; /* u64 because REQ_DATA_LEN is express in byte not in bit */
    u32 RESP_CAN_ID;
};

```

5.1.3. Data structure for REQUEST Do CAN (ISO 15765-4) protocol parameter

```

/* Data structure for DoCAN Request parameters */
struct RequestDoCanParameter
{
    u32 REQ_CAN_ID;
    u8 ID_PROTOCOL;
    enum Can_ID_format CAN_ID_FORMAT;
    u8 REQ_DATA_LEN;
    u8 REQ_DATA[8];
    u32 RESP_CAN_ID;
    u8 TA_LSB_PRESENT;
    u8 REQ_REPLY_LEN;
};

```

5.1.4. Data structure for REQUEST Volkswagen protocol parameter

```

struct RequestVWParameter
{
    u8 ID_PROTOCOL; Identifier of the Diagnostic Protocol;
    u8 CAN_ID_LGC; Logical ID of the target;
    u8 REQ_DATA_LEN; length of REQUEST DATA;
    u8 REQ_DATA[8]; Request Data, content of the request message payload
};

```

5.1.5. Data structure for REQUEST OPEL protocol parameter

```

struct RequestOPELParameters
{
    u8 ID_PROTOCOL;
    u16 REQ_CAN_ID; CAN ID of the requests to be sent;
    u8 REQ1_DATA_LEN; length of 1st REQUEST DATA;
    u8 REQ1_DATA[8]; Request Data, content of the 1st request message payload;
    u16 RESP1_CAN_ID; ID of the CAN message received as response to the 1st request;
    u8 REQ2_DATA_LEN; length of 2st REQUEST DATA;
    u8 REQ2_DATA[8]; Request Data, content of the 2nd request message payload;
    u16 RESP2_CAN_ID; ID of the CAN message received as response to the 2nd
request;
};

```

5.1.6. Data structure for REQUEST MINI protocol parameter

```

struct RequestMiniParameters
{
    u8 ID_PROTOCOL;
    u16 REQ_CAN_ID; ID of the request CAN messages;
    u8 PREREQ_DATA_LEN; length of PREREQ_DATA;
    u8 PREREQ_DATA[8]; Pre - Request Data, content of the 1st request message payload;
    u16 RESP_CAN_ID; ID of the CAN messages received in response;
    u8 REQ_DATA_LEN; length of REQUEST DATA;
    u8 REQ_DATA[8]; Request Data, content of the 2nd request message payload;
};

```

5.1.7. Data structure for OPERATION reading mode parameter

```

struct OperationDecodeParameter
{
    u16 PAR_ID;
    u8 DPRG_SOURCE;
    u8 NUM_TOT_STEPS; total number of steps to perform the calculation;
    struct OperationStepDecode INPUTS_OP_STEPS_X[TRL_NUM_TOT_STEPS]; number of inputs
of step \#x;
    struct ValueEncoding VALUE_ENCODING;
};

struct OperationStepDecode
{
    u8 NUM_OPS; numbers of operations to be done;
    u8 NUM_INPUT;
    u16 INPUTS_PARS_ID[TRL_OP_NUM_INPUTS];
    u8 OP_ID[TRL_OP_NUM_OPS];
};

```

Furthermore the data structure related to DCF-TRL interface can be describe as following

```
enum TrlResult
{
    TRL_RESULT_SUCCESS = 0,
    TRL_RESULT_GENERIC_FAIL
};
```

Most API calls have a return type of TrlResult to tell if the call succeeded. The caller is required to always check the result.

```
enum TrlInternalEvent
{
    TRL_EVENT_CAN_BUSCONF_COMPLETE,
    TRL_EVENT_CAN_BUSCONF_FAIL,
    TRL_EVENT_VIN_READ_COMPLETE,
    TRL_EVENT_VIN_READ_FAIL,
    TRL_EVENT_VDD_READ_COMPLETE,
    TRL_EVENT_VDD_READ_ERROR,
    TRL_EVENT_ALARM,
    TRL_EVENT_ERROR
};
```

This is used to signal the completion of an operation to the DCF.

```
typedef u16 TrlExternalEventType;
```

Used by the DCF to signal an external event

```
struct TrlVinLoadResultRow
{
    u8 vinValue[MAX_STRING_DATA_SIZE];
};

struct TrlVinLoadResult
{
    struct TrlVinLoadResultRow results[MAX_VIN_NUM];
    u8 resultsLen;
};
```

This contains the VIN values that have been read from the bus. The values have to be verified

```
enum TrlState
{
    TRL_STATE_RESET,
    TRL_STATE_WAITING_BUSCONFIG,
    TRL_STATE_AUTOCONFIGURING_BUS,
    TRL_STATE_WAITING_VINREAD,
    TRL_STATE_READING_VDD,
    TRL_STATE_READING_VIN,
    TRL_STATE_WAITING_VDD,
    TRL_STATE_IDLE,
    TRL_STATE_RUNNING,
    TRL_STATE_ERROR
};
```

This data structure describe the current operation in execution by the TRL.

5.2. Software Interfaces

The TRL software is designed to run as a state machine see the following TRL State Machine. This state machine is made by up to ten states that define at each time the current operation is running by the TRL software.

The TRL implementation provides a set of interfaces that are called to configure and activate the requirement functionalities.

In the following the description of functions interface related to my work will be done in order to assert the functionalities of the TRL software.

These interfaces can be describe either at block level with others block or between the TRL and others module in the design.

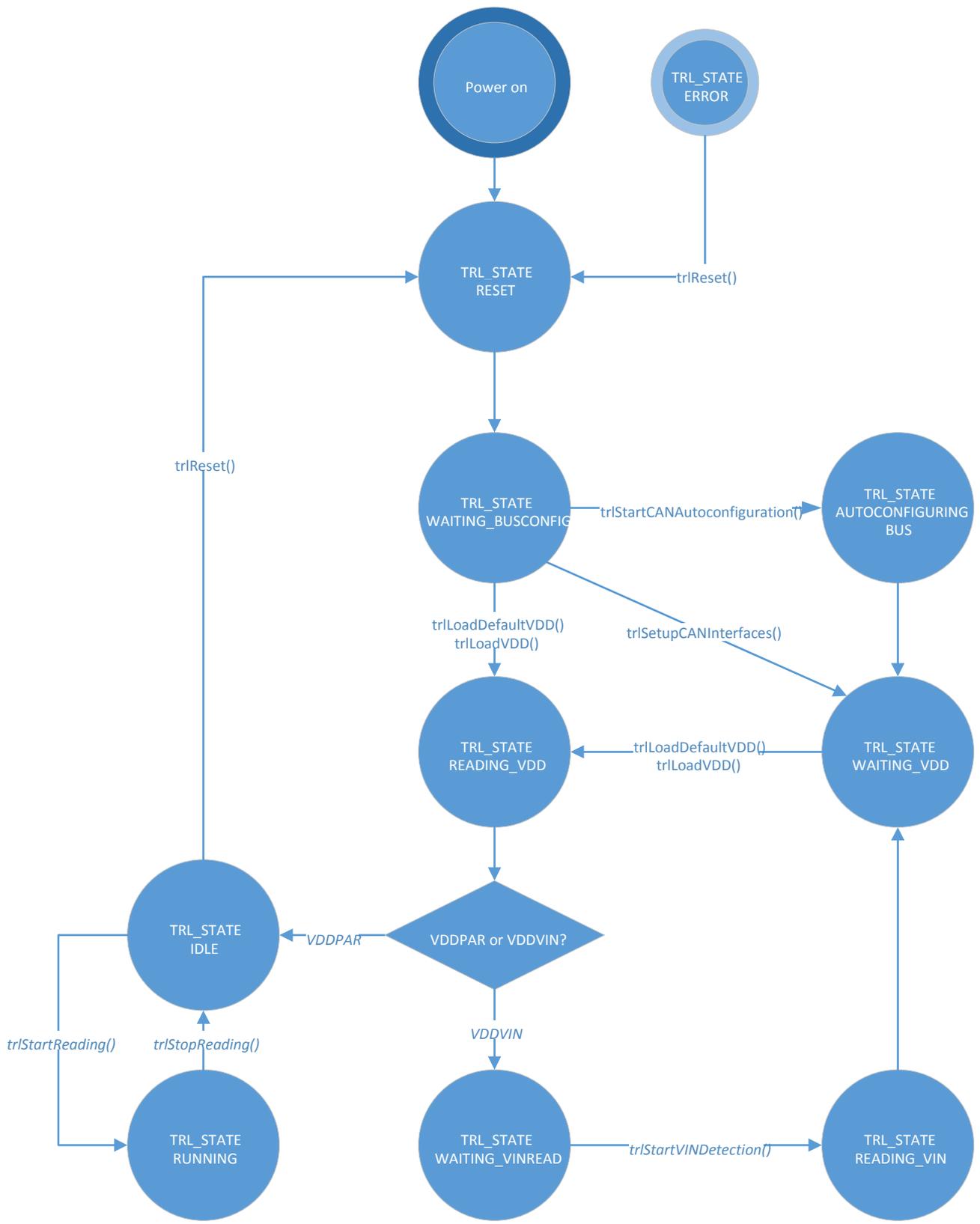


Figure 12: TRL State Machine

5.2.1. DCF-TRL Interface functions

In the following the interface and its relative call-back that describe the state machine of the TRL above will be presented.

These are followed by a comment that possibly states the role of that interface.

```
void TrlReset(void);
```

Used to return the TRL in the default state

```
enum TrlResult trlSetupCallbacks(TrlCallbackEvent, TrlCallbackParameter, TrlCallbackAlarm);
```

Setup the callbacks

```
enum TrlResult trlSetupCanInterfaces(struct TrlCanInterfacesConfig *);
```

Used to load the can interfaces configuration

```
enum TrlResult trlStartCanAutoconfiguration();
```

Starts the automatic configuration of the can interfaces, the result will be returned in a callback

```
enum TrlResponse trlLoadVdd(tU8 * vddData, tU32 vddLen);
```

sends the pointer to the memory containing the vdd file to be processed. The vdd file can be a PAR or a VIN

```
enum TrlResult trlStartVinDetection(void);
```

Starts a VIN read operation. The result will be returned in a callback

```
enum TrlResult trlStartReading(void);
```

Enables processing of the data from the CAN bus

```
enum TrlResult trlStopReading(void);
```

Temporarily stops the processing of data from the CAN bus

```
void trlNotifyExternalEvent(TrlExternalEventType);
```

Used to notify an external event

```
enum TrlResult trlRequestParameterValue(TrlParamIdType par_ID);
```

Nonblocking request of one parameter from the can bus. The result will be returned in a callback.

```
void trlRequestSavedParameterValue(TrlParamIdType par_ID);
```

Used to read immediately the value of a parameter. The parameter is retrieved from the internal database.

```
enum TrlState trlGetCurrentState();
```

Gets the current state of the TRL

5.2.2. Callbacks

By definition a call-back function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

This Call-back are function pointer executed asynchronously with respect to each state.

```
void(*TrlCallbackParameter) (TrlResult result, TrlParamIdType par_id, struct TrlParameterValue *val);
```

Called when the TRL wants to notify a new value of a parameter

```
typedef void(*TrlCallbackEvent) (enum TrlEvent, void *eventData, tU16 len);
```

An internal event was generated by the TRL

5.2.3. VDD DECODER INTERFACE

This interface is at block level as describe the architecture design of the TRL , and has the main task of implementing the logic that allow the TRL from the loaded VDD file , first read the past buffer in order to peek the protocol type that will be used for requesting the diagnostic parameter then based on that protocol type , trigger the parsing block to load the corresponding data structure and the finally prepare the parameter ready to be use by the data manager , by configuring the data structure in the data manager.

The interface of the block interface towards others block is

```
enum TrlResult vddDecodeBuffer(struct BufferReaderState *state, struct VddlParameterConf *parameter);
```

The relative sub-function this block rely on are specifies with the following interfaces :

```
enum TrlResult vddlPeekProtocolType (const struct BufferReaderState *_state, enum VddlProtocolType *protoType)
```

This interface allow to read the protocol type to use for the request of the diagnostic parameter in other to select the right data structure to load.

```
enum TrlResult vddlPrepareOperationParameters(struct  
OperationDecodeParameter *parameters, struct VddlParameterConf  
*ConfParameters)
```

```
enum TrlResult vddlPrepareEobdParameters(struct RequestOBD_EOBDParameter  
*params, struct VddlParameterConf * parametersConf)
```

```
enum TrlResult vddlPrepareOPELParameters(struct RequestOPELParameters  
*params, struct VddlParameterConf * parametersConf)
```

```
enum TrlResult vddlPrepareMINIParameters(struct RequestMiniParameters  
*params, struct VddlParameterConf * parametersConf)
```

```
enum TrlResult vddlPrepareVWParameters(struct RequestVWParameter *params,  
struct VddlParameterConf * parametersConf)
```

```
enum TrlResult vddlPrepareDoCanParameters(struct RequestDoCanParameter  
*params, struct VddlParameterConf * parametersConf)
```

```
enum TrlResult vddlPrepareListenParameters(struct CanListenParameters  
*params, struct VddlParameterConf * parametersConf)
```

5.2.4. VDD PARSER INTERFACE

This interface is at block level as describe in the architecture design of the TRL , and is able to parse the VDD file based on a specific protocol and to load the data structures that will be useful by order block in the design.

In particular it received as input a buffer from the decoder block containing the remaining bytes from VDD file and then extract the corresponding byte field and load the corresponding protocol based data structure.

The different interface for decoding a parameter are described as following :

```
enum TrlResult vddlDecodeListenParameters(struct BufferReaderState
*state, struct CanListenParameters *params)
```

This interface allows to load a listen based diagnostic parameter data structure.

```
enum TrlResult vddlDecodeOperationParameter(struct BufferReaderState
*state, struct OperationDecodeParameter *params)
```

This interface allows to load an operation based diagnostic parameter data structure.

```
enum TrlResult vddlDecodeMINIParameter(struct BufferReaderState *state,
struct RequestMiniParameters *params)
```

This interface allows to load a Mini based diagnostic parameter data structure.

```
enum TrlResult vddlDecodeOPELParameter(struct BufferReaderState *state,
struct RequestOPELParameters *params)
```

This interface allows to load Opel based request diagnostic parameter data structure.

```
enum TrlResult vddlDecodeVWParameter(struct BufferReaderState *state,
struct RequestVWParameter *params)
```

This interface allows to load VW based request diagnostic parameter data structure.

```
enum TrlResult vddlDecodeOBD_EOBDParameter(struct BufferReaderState
*state, struct RequestOBD_EOBDParameter *params)
```

This interface allows to load OBD/EOBD based request diagnostic parameter data structure.

```
enum TrlResult vddlDecodeDoCanParameter(struct BufferReaderState *state,
struct RequestDoCanParameter *params)
```

This interface allows to load DoCan based request diagnostic parameter data structure.

5.2.5. Operation Manager interface

This block as describe in the architecture design , is responsible for the evaluation a parameter of type Operation in reverse polish notation and then responsible of performing a logic or mathematical operation.

In particular it receive as input the decoded operation parameter then :

1. create the operation by internally transformed the parameter operation into reverse polish notation using an internal stack memory and insert the transform expression to it internal data base. For this purpose it implement the following interface.

```
tr10mCreateOperation(struct OmOperationExpression* opDecodeParameter)
```

2. Execute the expression transformed in RPN
for this purpose it implement the following interface

```
tr10mExecuteOperation(TrlHandle opIndex, struct OperationDependencyValues *depValues,  
struct OperationResult* outResult);
```

Moreover it implement other interface as :

```
void tr10mReset(void)
```

For reset the internal data base

```
static void tr10mInitializeOpScratchpad(u32 handle)
```

To initialize the internal data base

```
void tr10mClearOperationData(void)
```

to clear a data expression previously initialized.

```
static enum TrlResult tr10mExecuteOperationSingle(struct OperationStack *operationStack,  
struct OperationOperatorScratchpad *operatorScratchpad);
```

This interface of executing the single operation type (e.g. SUM, MAX, MULT , DIV , etc..).

5.2.6. VIN READER interface

This block implements the algorithms needed to read the VIN number Figure 7 by using hardcoded data or the Vehicle Diagnostic Description file received from the Data Collection Firmware.

This block implement the following interfaces :

```
enum TrlResult v1StartReading(u8 _tryDifferentSpeeds);
```

that is responsible to start the VIN reading operation by trying vary speed bus type and when the speed is well setup the actual reading occur through the following interface

```
static void requestVINbyIndex(u16 index);
```

by configuring the HLD with the current speed bus through the following interface.

```
tr1HldConfigureBus(0, busSpeeds[speedIndex], 11);
```

Moreover when the reading of VIN succeed , the VIN is saved to the memory through the following interface

```
static void saveVinDataByIndex(u16 index);
```

5.2.7. OBDII/EOBD interface

This as describe in the design architecture is specific to the OBDII/EOBD standard and is responsible of implementing the OBDII/EOBD protocol for diagnostic parameter request on the CAN bus.

In order to perform it job it implement the following interfaces :

```
struct TrlResultExt trlHldEobdConfigureParameter(const struct TrlHldConfigEOBD* config);
```

that is responsible to configure TRL-HLD with the type of parameter request based on the corresponding decoded data structure.

After the parameter is configured , the connection is created for communication over the CAN bus based on the ISO-TP standard through the following interface :

```
struct TrlResultExt isoTpCreateConnection(struct IsoTpConnectionParameters* config);
```

Once the connection has been well setup an event is generated for waiting the response from the request through the following interface :

```
void trlEobdIsoTpEventCB(TrlHandle isoTpHandle,struct IsoTpConnectionParameters  
*connectionParameters, enum TrlHldResult error);
```

Afterwards when a new message is present on the bus it's retrieved calling the following interface call-backs

```
void trlEobdIsoTpMessageCB(TrlHandle isoTpHandle,struct IsoTpConnectionParameters  
*connectionParameters, const u8 * data, u16 dataLen)
```

In particular this callback is responsible of passing the respond message request to the TRL-CORE through the TRL-HLD INTERFACE namely

```
void trlHldNewMessageCallback(enum TrlHldLibId libId, TrlHandle handle,const u8 * data, u16  
dataLen
```

Chapter 6.

Experimental test and results

In this chapter will be describes vary test activities performed in order to verify and validate the main requirement functionality of the TRL based on specific examples.

In particular The testing activity is executed with two different approaches:

- ✓ Software tests on PC-based simulation environment to check the main functionalities of the TRL.
- ✓ Testing executed on the hardware plate form evaluation board a couple with CAN bus protocol management tool (CANalyzer)

6.1. Test on PC based Simulated Environment

The software test simulation has been performed on the PC running window 7 64 bit operating system with visual studio version 2017 installed.

6.1.1. Test plan

The simulation test is like a functional test where each functionality of the TRL software has been tested. The environment test is like an automated test and it's compose of several Item test case that when launched simulate the whole system at once and produce a global Success/failure result.

The Test case in our case are describes as Item function that implement the test of a single request protocol or the test of a single block module made by many functions or test of sets of block modules including complex functions or finally the test a whole system including all functions and modules of the TRL.

6.1.2. Test Approach

The steps for testing an item are

- ✓ Write the test case for that specific item
- ✓ Insert the test case in the test list queue
- ✓ Run the automated test environment

- ✓ Observe results test

6.1.3. Test description flow

The main modules/files that compose the simulated test environment are described as following

- ❖ The Main.cpp file , that contains the Main function and the RunTest function (see the following fig) .
 1. The Main function initiate the automated simulation test by triggering the RunTest function and at the end records the global success of the test.
 2. The RunTest function which it role is to get all test case through the getTestList function (see the following fig) , then execute them and finally records result of each specific test case in the log file result.
- ❖ The TestList.cpp file which contains the GetTestList function , and the definition of each test case function.
 1. The GetTestList function has a role of declaring each test case .
- ❖ The testEngine.cpp file which contains interface function for simulating the behaviour of the engine. It represented following figure by the Engine module
- ❖ The source file folder that define the TRL made by Trl_core and Trl_hld modules.
- ❖ Test result log file that registers all test results either for a specific test case or for the global result of the test.

The following figure describe how the function modules composing the simulated software environment test are interconnected and the steps to perform the automated test simulation .

SOFTWARE TEST DESCRIPTION FLOW

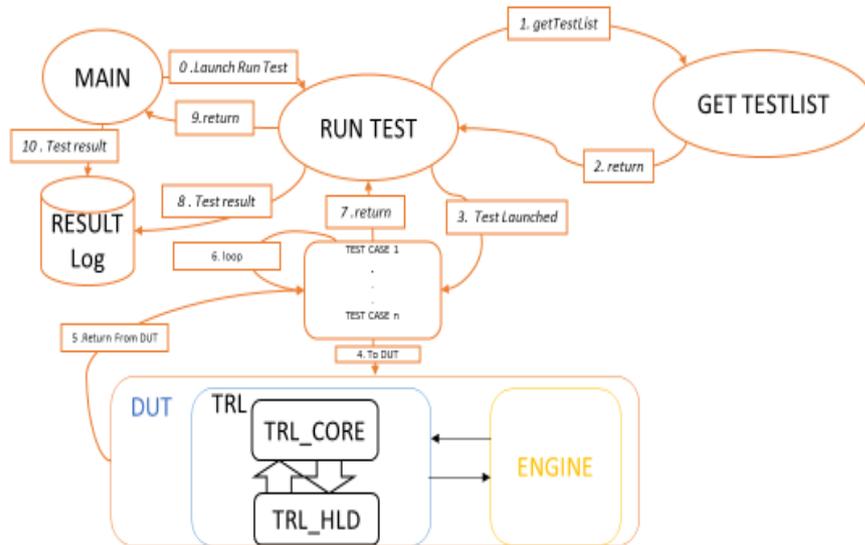


Figure 13: Environment test

6.1.4. Test Results

6.1.4.1. Test definition

Test are defined according to the following scenarios:

- System test: test of the whole TRL functionalities
- Module test: test of single block functionality
- Specific feature test: test of complex operation or specific function
- Communication protocol test: test of single protocol

In the following test result obtained when simulating the TRL software, will be specify for each category of test for one specific diagnostic parameter.

6.1.4.2. System test

Id test case	Name \Description test case	Input	Execution steps	Expected Result
TC	testListenONCan: check of listen CAN (Vehicle: Abarth 595 106kW year 2016)	Session of the byte sequence in VDD file	<ol style="list-style-type: none"> 1. Decode of vdd input data parameter according to the CAN RAW protocol and prepare the parameter to be loaded into the HLD 2. Loading vdd decoded parameter for HLD configuration 3. Listen on the can bus the the expected response 4. Parse the response for obtaining the expected value 5. Check for the rightness of the expected value 	Parameter type: Steering Wheel Angle Value: +360°

Table 3: system test

6.1.4.3. Module test

Id test case	Name \Description test case	Input	Expected Result	
TC	testSimpleOperation: check of different type of operation	Buffer value contening operand for the operation	<ol style="list-style-type: none"> 1. Configure the data structure for CAN raw 2. Configure HLD and create CAN raw parameters 3. Configure the operation manager for operation type max 4. Create the configure operator parameter 5. Listen on the bus set of data from engine 6. Read values from bus and check each time the current maximum value 	The current maximum value

Table 4: Module Test

6.1.4.4. Specific feature test

Id test case	Name \Description test case	Input	Execution steps	Expected Result
TC	testOperationDEBOUNCE: check of vehicle diagnostic parameters, based on a calculation of DEBOUNCE that uses one or more inputs	1. PAR_ID of the parameter in observation 2. Time T1 3. Time T2	1. Definition of a parameter with type operation and operator name DEBOUNCE 2. Create a parameter 3. Execute the operation with different unsigned handle value (par_id, t1,t2)	0 : DEBOUNCE PAR OUTPUT LOW -1 : DEBOUNCE PAR OUTPUT HIGH

Table 5: Feature Test

6.1.4.5. Communication protocols test

Id test case	Name \Description test case	Input	Expected Result
TC	testEobdRequest: check of EOBD	Parameter for configuration	1. Configuration of the HLD for EOBD request protocol Success (value is 1)

	request with a single parameter	of the channel: CAN_CH REQ_ID RES_ID REQ_MODE RDI	2. Update the request with the ID and create a connection 3. Listen configuration data from TRL 4. Sending of the response by the engine 5. Check of the rightness of the request	Unsuccess (value is 0)

Table 6:
Protocol
test

6.2. Hardware Test

6.2.1. Hardware setup

The test performed in hardware have been executed with the following setup embedded system environment see the following pictures

- ❖ CAN bus protocol management software tool (CANalyzer)
- ❖ CANcaseXL hardware
- ❖ CAN bus wire
- ❖ The Teseo III embedded plate form evaluation board microcontroller STA8090FG , having core ARM946
 - free CPU resources: max 10% of the total (total is 200MIPS)
 - available RAM space: max 30KB
 - available FLASH space (for code): max 450KB
 - C-language source code, compiled with ARM DS5

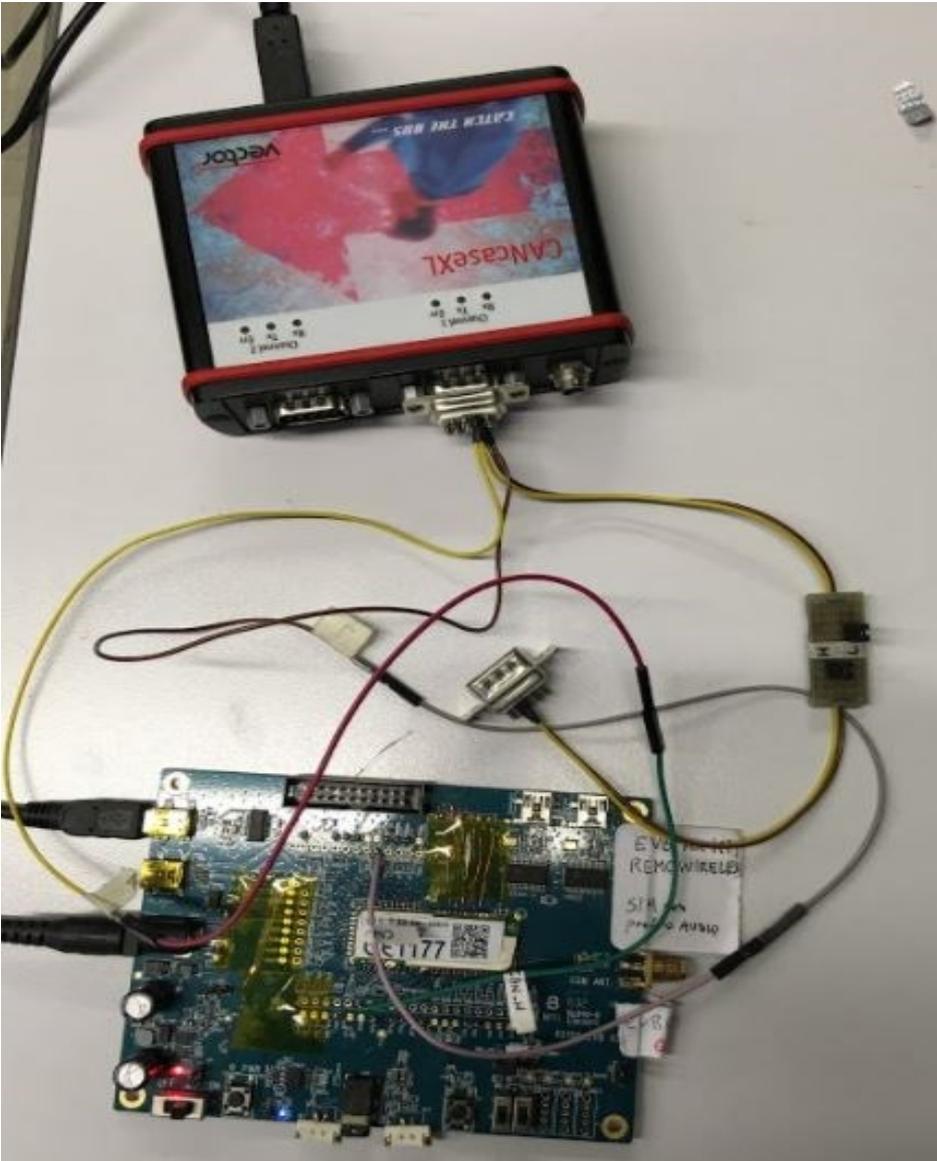


Figure 14 : Hardware testing

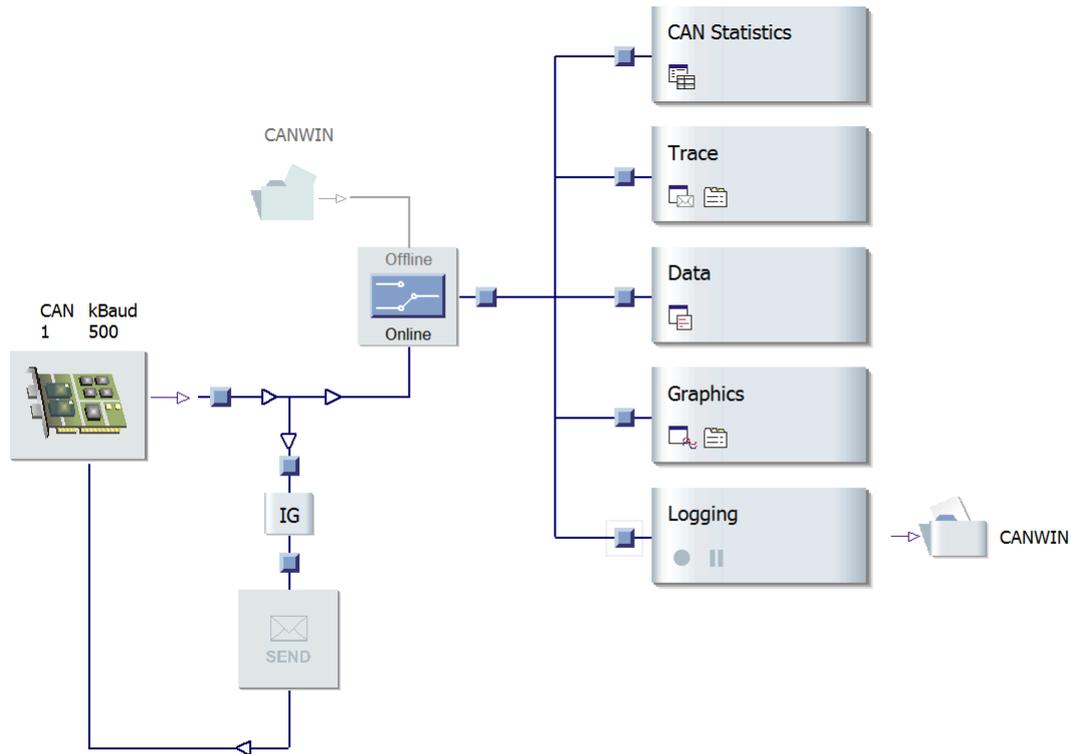


Figure 15: Environment Tool setup

6.2.2. Test description Approach

The approach used to perform test in hardware is to program a Can bus management tool in order to behave like an engine for providing requested diagnostics parameters to the Translation library firmware.

In particular as the Translation library firmware run as a state machine , it's also configured in order to perform request for interested diagnostics parameters.

6.2.2.1. Configuration of the TRL

The configuration of TRL is done by an external Data Collection Firmware through the DCF-TRL interface.

In the following is showed how the TRL is configured for requesting a given diagnostic parameter. The configuration is done for request on the vehicle **MERCERDES VE_PLAT_MRC01_** Plate form.

➤ Create Parameter Vin Reading

```
configParameter.encoding = DATA_ENCODING_STRING;
configParameter.parameterId = 0x2000;
configParameter.finalSizeBytes = 17;

decodeString.splitParameters.endianness = DATA_ENDIANNESS_LITTLE;
decodeString.splitParameters.lengthBits = 17*8;
decodeString.splitParameters.startBit = 8;

    config.CAN_CH = 0;
    config.REQ_ID = 0x7E0;
    config.REQ_MODE = 0x09;
    config.RDI = 0x02;
    config.RES_ID = 0x7E8;
    config.addressMode = TRL_CAN_ADDR_DEFAULT;

trlHldConfigureParameter(TRL_HLD_LIBID_EOBD, configurationHandle, &config, sizeof(config))

dmCreateParameter(configurationHandle, &configParameter, &decodeString);
```

➤ Create Parameter Fuel Level Absolute

```

configParameter.encoding = DATA_ENCODING_INTEGER_UNSIGNED;
decodeParameter.rawEncoding = DATA_ENCODING_INTEGER_UNSIGNED;
configParameter.parameterId = 0x201B;
configParameter.finalSizeBytes = 2; //in 0.1 litri

```

```

struct TrlHldConfigEOBD configEOBD;

```

```

        decodeParameter.gain    = 10;
        decodeParameter.offset  = 0;
        decodeParameter.splitParameters.endianness =
DATA_ENDIANNESSEBIG;
        decodeParameter.splitParameters.startBit    = 0;
        decodeParameter.splitParameters.lengthBits  = 8;

        configEOBD.CAN_CH      = 0;
        configEOBD.REQ_ID      = 0x7E0;
        configEOBD.REQ_MODE    = 0x22;
        configEOBD.RDI         = 0x6038;
        configEOBD.RES_ID      = 0x7E8;
        configEOBD.addressMode = TRL_CAN_ADDR_DEFAULT;

```

```

trlHldConfigureParameter(TRL_HLD_LIBID_EOBD, configurationHandle,
&configEOBD, sizeof(configEOBD));

```

```

dmCreateParameter(configurationHandle, &configParameter,
&decodeParameter);

```

```

vdd1LoadDCFVDD :

```

```

    CreateParameterOperationFuelLevelAbsoluteMIN();

```

Once the parameter has been configured , the TRL is let free to interact with the engine through the Can Bus.

6.2.2.2. Setup of Can bus management tool for providing requested diagnostic parameter

The Engine

In the following is shown the picture that describe how to CancaseXL hardware which behave as the engine, is program in to respond in the right way when received a request for a diagnostic parameter in this case the **fuel absolute level** parameter.

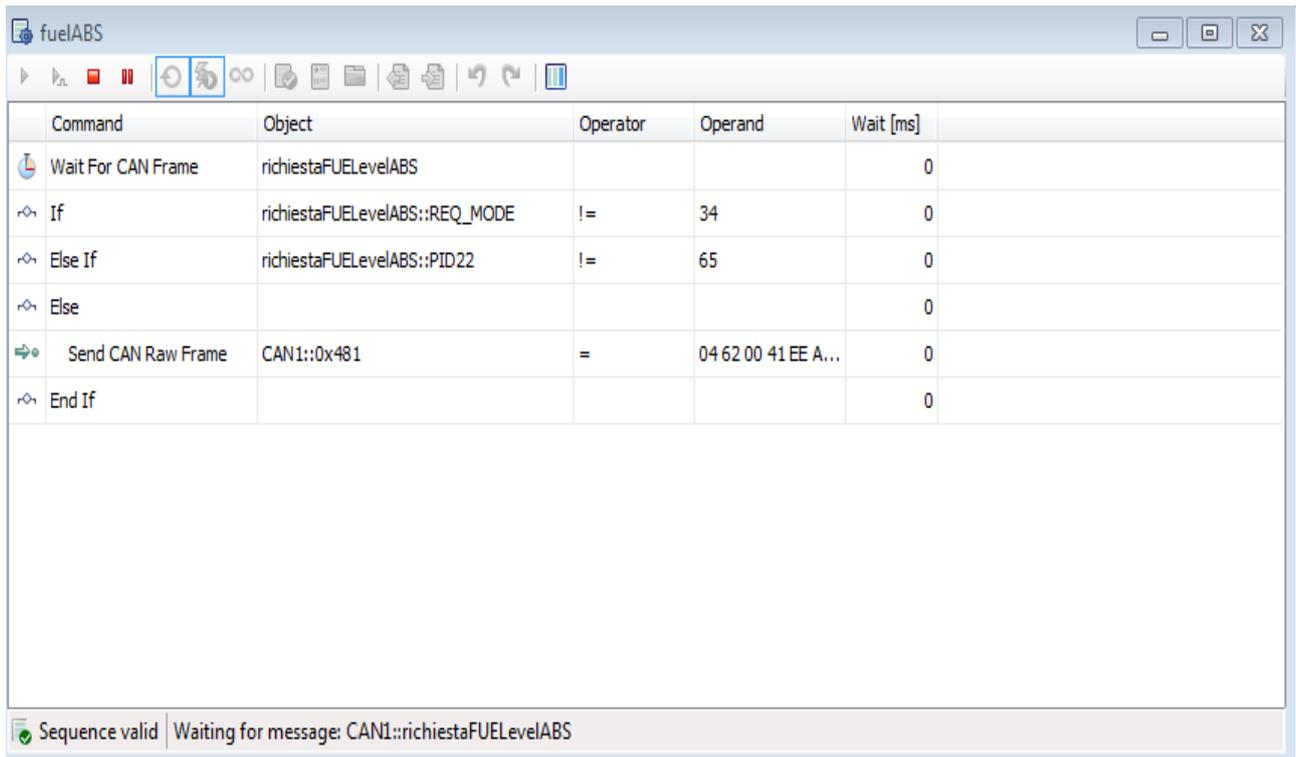


Figure 16: Diagnostic parameter response

The following figure list some set of diagnostics parameters among all, the engine is able to respond from a request.

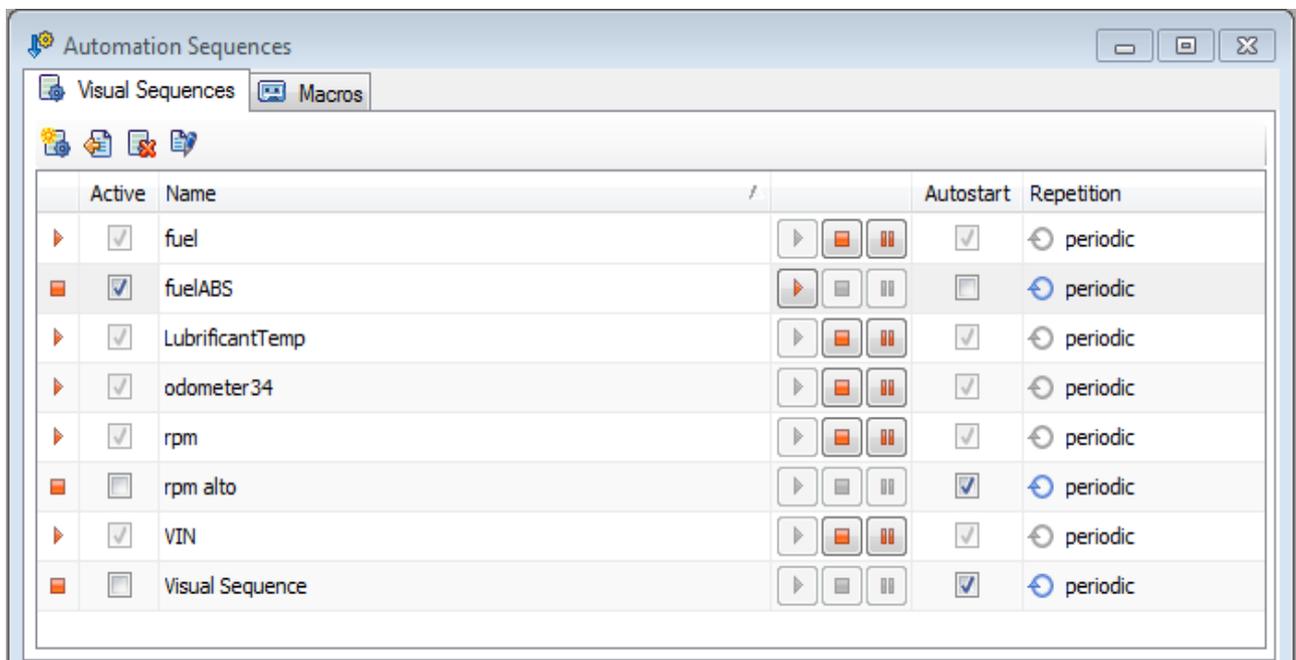


Figure 17: Parameter request

The following picture is trace over the CAN bus for parameter request and response data.

Time	Chn	ID	Name	Event Type	Dir	DLC	Data
00:00:00							
1010.001387	CAN 1	19F		CAN Frame	Tx	8	00 00 00 00 00 02 10 00
1010.231159	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 05 00 00 00 00 00
1010.621139	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 05 00 00 00 00 00
1011.001533	CAN 1	19F		CAN Frame	Tx	8	00 00 00 00 00 02 10 00
1011.031118	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 05 00 00 00 00 00
1011.441122	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 42 00 00 00 00 00
1011.841082	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 42 00 00 00 00 00
1012.001276	CAN 1	19F		CAN Frame	Tx	8	00 00 00 00 00 02 10 00
1012.231068	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 42 00 00 00 00 00
1012.651046	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 42 00 00 00 00 00
1013.001346	CAN 1	19F		CAN Frame	Tx	8	00 00 00 00 00 02 10 00
1013.061033	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 42 00 00 00 00 00
1013.451015	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 42 00 00 00 00 00
1013.870993	CAN 1	60A	richiestaFUELevelABS	CAN Frame	Rx	8	03 22 00 41 00 00 00 00
1013.871819	CAN 1	481	risposta2	CAN Frame	Tx	8	04 62 00 41 EE AA AA AA
1014.001299	CAN 1	19F		CAN Frame	Tx	8	00 00 00 00 00 02 10 00
1014.170985	CAN 1	682		CAN Frame	Rx	8	03 22 03 01 00 00 00 00
1014.570979	CAN 1	682		CAN Frame	Rx	8	03 22 03 01 00 00 00 00
1014.980979	CAN 1	682		CAN Frame	Rx	8	03 22 03 01 00 00 00 00
1015.001277	CAN 1	19F		CAN Frame	Tx	8	00 00 00 00 00 02 10 00
1015.380965	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 21 00 00 00 00 00
1015.780955	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 21 00 00 00 00 00
1016.001573	CAN 1	19F		CAN Frame	Tx	8	00 00 00 00 00 02 10 00
1016.180955	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 21 00 00 00 00 00
1016.590929	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 05 00 00 00 00 00
1016.980889	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 05 00 00 00 00 00
1017.001293	CAN 1	19F		CAN Frame	Tx	8	00 00 00 00 00 02 10 00
1017.380885	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 05 00 00 00 00 00
1017.790845	CAN 1	7E0	richiestaFUEL	CAN Frame	Rx	8	02 09 02 00 00 00 00 00
1017.791513	CAN 1	7E8	risposta1	CAN Frame	Tx	8	05 62 10 0C 01 01 AA AA
1017.820799	CAN 1	7E0	richiestaFUEL	CAN Frame	Rx	8	02 09 02 00 00 00 00 00
1017.821519	CAN 1	7E8	risposta1	CAN Frame	Tx	8	05 62 10 0C 01 01 AA AA
1017.850800	CAN 1	7E0	richiestaFUEL	CAN Frame	Rx	8	02 09 02 00 00 00 00 00
1017.851519	CAN 1	7E8	risposta1	CAN Frame	Tx	8	05 62 10 0C 01 01 AA AA
1017.890842	CAN 1	7E0	richiestaFUEL	CAN Frame	Rx	8	03 22 22 2E 00 00 00 00
1017.891480	CAN 1	7E8	risposta1	CAN Frame	Tx	8	05 62 10 0C 01 01 AA AA
1017.940859	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 0D 00 00 00 00 00
1018.001344	CAN 1	19F		CAN Frame	Tx	8	00 00 00 00 00 02 10 00
1018.310828	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 0D 00 00 00 00 00
1018.720808	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 0D 00 00 00 00 00
1019.001212	CAN 1	19F		CAN Frame	Tx	8	00 00 00 00 00 02 10 00
1019.120811	CAN 1	7DF	richiestaDiagnostica	CAN Frame	Rx	8	02 01 1C 00 00 00 00 00

Figure 18:Can bus trace

Chapter 7.

Translation Library Software Verification with Polyspace Tool

7.1. TRL verification with Polyspace Code Prover And MISRA C check rules

This section of work will mainly describe the verification result that have been done on the TRL software and the compliance with MISRA C 2012 rules standard.

7.1.1. Project Setup

Creating a project in polyspace code prover consist first to choose the project name then locate all source files and includes files to be added to the project in their proper folders name.

The figure bellow summarize how the created project in polyspace code prover look like and list different folders containing proper source files.

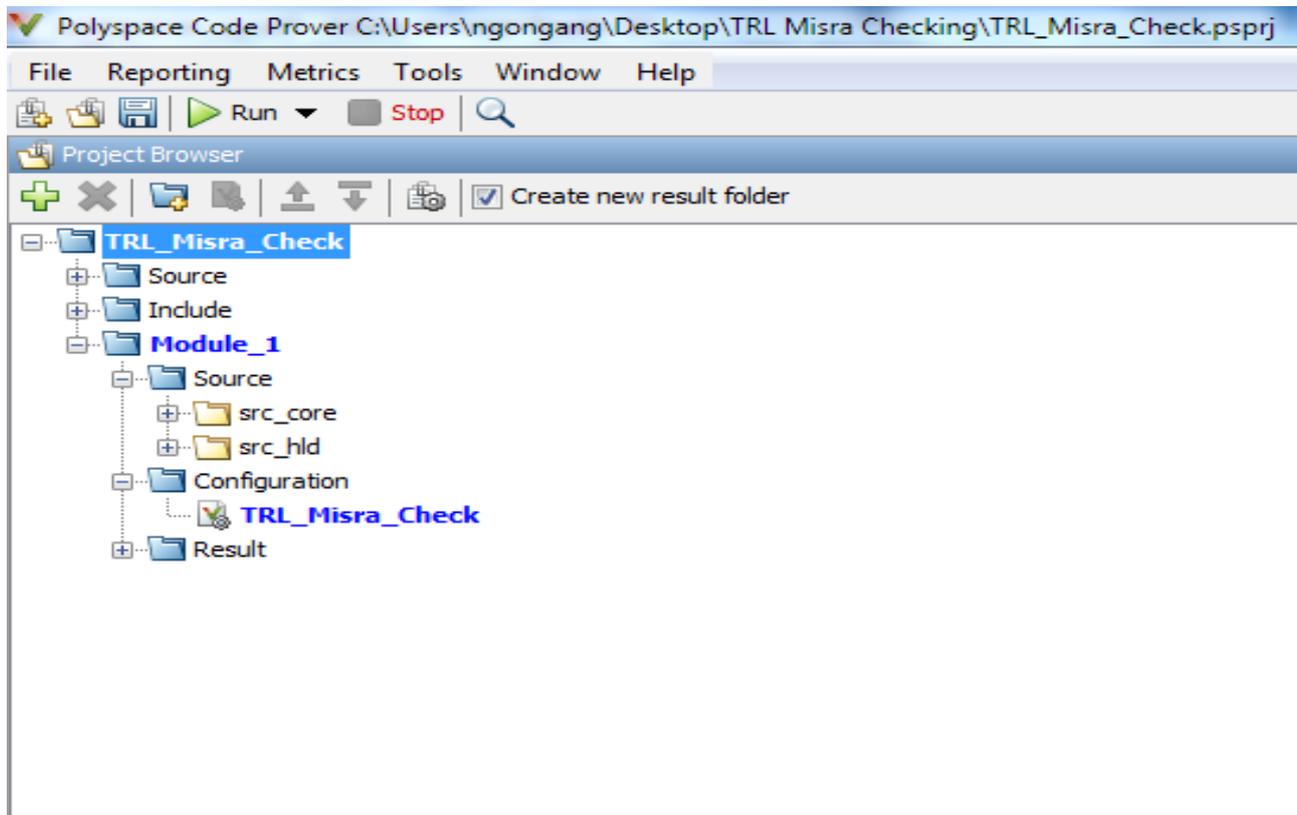


Figure 19 : Polyspace code prover file organization interface

7.1.2. Project configuration : Coding Rule and Metrics

The TRL project has been configured in polyspace code prover software to use C language as target language. Moreover the project have been configured to check MISRA C 2012 rules compliant with the ISO 26262 standard guideline and have enabled to compute code metrics.

7.2. Results Analysis

This paragraph will mostly presenting results obtains when performing Misra Rules checking for the safety compliance with ISO 26262.

The TRL software verification has been run with Polyspace code prover in order to check MISRA C coding rules and quickly identify and fix obvious defects due to run time error. The methodology adopted during verification has been that of modular verification based on either each component verification then sets of component and finally on the overall design. The verification process technique is like a processor-in-the-loop (PIL) where the target processor were a configured ARM like processor architecture . The verification has been run in the loop and based on generated results bugs has been progressively fixed .

Target & Compiler	
Target Language	
Source code language	C
Target Environment	
Target operating system	Linux
Target processor type	ARM_like <input type="button" value="Edit"/>
Dialect	iso
<input type="checkbox"/> Respect C90 standard	
Compiler Behavior	
<input type="checkbox"/> Division round down	
Enum type definition	defined-by-standard
Signed right shift	Logical

Figure 20 : Target processor

As the verification has been setup to run in modular form, the following figure shows the verification running on three files component belong to the TRL core. The summary graph report result in it higher part shows that these modules are free from run time error (red color). Moreover in the other hand from the MISRA C check part it shows categories of rule that have been violated and how many time they have been violated in the verification process.

Results Summary			
Checks & Rules			
Family	Information	File: (3)	
- Run-time Check		1	16 752
+ Gray Check		1	
+ Orange Check		16	
+ Green Check			752
- MISRA C:2012			7
+ 4 Code design			1
+ 9 Initialization			1
+ 10 The essential type model			2
+ 11 Pointer type conversions			2
+ 18 Pointers and arrays			1

Figure 21: Coding color summary

However by running the verification process on the entire TRL software , the following graph from result metrics indicate that the verification on the software have been all covered either or in term of function or in term of operation in the code.

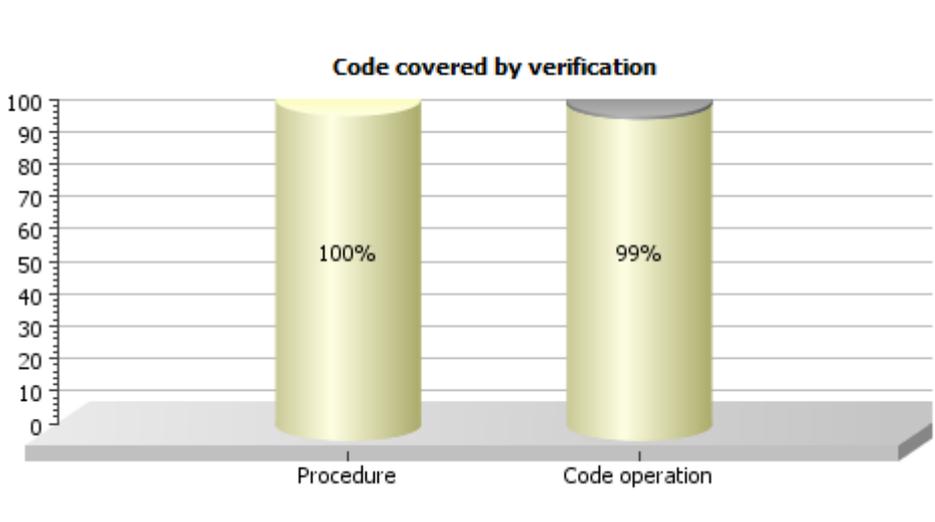


Figure 22:TRL Coverage

Moreover the following distribution assert the level of safeness of the software with 94% of proven coding rules checked.

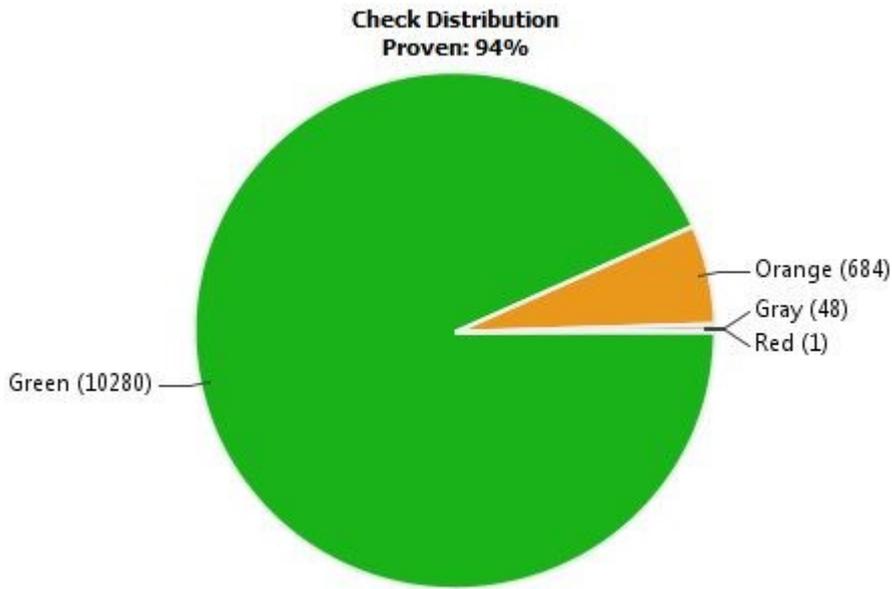


Figure 23: Distribution Check

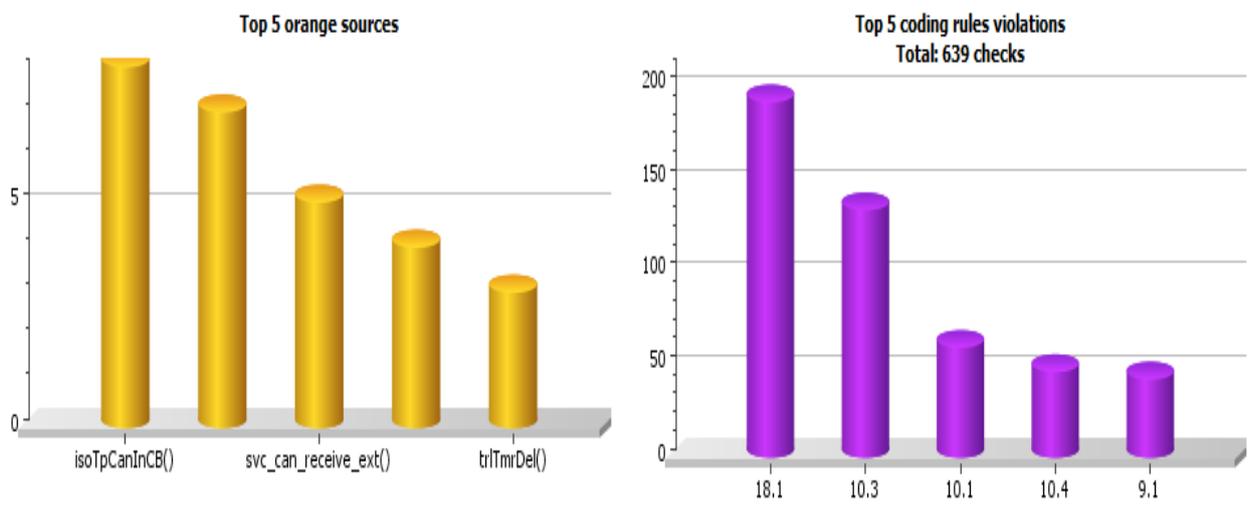


Figure 24: Coding Rule violated

Chapter 8.

Conclusion and future work

In conclusion this thesis has proposed a design architecture and implementation of an embedded Translation library firmware which can be integrated into traditional vehicle On-Board device for requesting diagnostics parameters over the can bus from large number of vehicles manufactures. Based on the information gathering on fields about the specific CAN message implemented by different vehicle manufactures of different brand and model, this translation library firmware can support multiple protocols request for diagnostics parameters namely OBDII/EOBD request protocol built upon the ISO ISO15765 standard; Diagnostics on Can request protocol (DoCAN) relying on ISO 15765-2 standard, Mini request protocol built upon ISO-TP, Volkswagen request protocol that rely on SAE J2819 TP 2.0 and finally CAN RAW and Opel protocol request that are custom implementation regardless of the vehicle model. Moreover, this thesis has also described the main software interface and data structure that allow the software to run. Furthermore, different approach of testing has been proposed either test based simulation on PC so-called software in the loop (SIL) or hardware test so called hard in the loop (HIL) have been adopted to check the correct behavior of the TRL software.

This TRL embedded firmware library is then able to request very large number of diagnostics parameters by means of a vehicle diagnostic description (VDD) file that is responsible for configuration of the firmware. These parameters can be from DTC code of system malfunction to VIN passing through engine rpm, tyre pressure, coolant temperature, lubricant temperature level and so many one. Afterwards the software implementation and testing, a static analysis verification with polyspace code prover tool of the TRL software have been performed in order to first let free the software from most critical run time error namely overflow, division-by-zero, out-of-bounds array access and secondly to check MISRA C coding rule compliance in order to guarantee the safety critical and security of the TRL software and the compliance with the ISO 26262 standard.

However the vary result obtained with polyspace code prover when running the static analysis verification both for "0" defects run time error and MISRA C rules check were not so highly optimal for instance as show on the graph above Figure 23: Distribution Check, we have obtain 94% code coverage proven and 5 category of MISRA C rule uncheck due to the limitation of time. This

obtained result is mainly because Polyspace tool didn't integrate earlier at the beginning of the development process of the TRL software.

Therefore due to the limitation of time as future work, a deeply investigation could be done by running the static analysis verification on each block component in the TRL design in order to get very high quality code and total compliance with the MISRA C coding guidelines.

List of figures

Figure 1: CanAlyzer tool	19
Figure 2 :T3 EVB	20
Figure 3: CanCaseXL	21
Figure 4 : OBU Design architecture	22
Figure 5: parameter ID	24
Figure 6 : VDD FORMAT.	25
Figure 7: VIN	26
Figure 8: TRL-Core	27
Figure 9 : TRL VIEW	30
Figure 10:DCF-TRL sequence diagram	31
Figure 11:VIN Detection sequence diagram	32
Figure 12: TRL State Machine	39
Figure 13: Environment test	48
Figure 14 : Hardware testing	52
Figure 15: Environment Tool setup	53
Figure 16: Diagnostic parameter response	56
Figure 17: Parameter request	56
Figure 18:Can bus trace	57

Figure 19 : Polyspace code prover file organization interface	58
Figure 20 : Target processor	59
Figure 21: Coding color summary	60
Figure 22:TRL Coverage	60
Figure 23: Distribution Check	61
Figure 24:Coding Rule violated	61

List of tables

Table 1 : Normative Standards	23
Table 2: Diagnostic Protocol	29
Table 3: system test	49
Table 4: Module Test	49
Table 5: Feature Test	50
Table 6: Protocol test	51

Bibliography

<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4519050>

http://web1.see.asso.fr/erts2010/Site/0ANDGY78/Fichier/PAPIERS%20ERTS%202010/ERTS2010_0009_final.pdf

<http://ldra.com/aerospace-defence/standards/misra-cc/>

<https://rmbconsulting.us/Publications/MisraC.pdf>

https://en.wikipedia.org/wiki/On-board_diagnostics

<https://www.sciencedirect.com/science/article/pii/S0950584903002076>

<https://www.ciklum.com/blog/key-embedded-software-trends-in-2017/>

<http://www.strategyr.com/pressMCP-7778.asp>

<http://ieeexplore.ieee.org/abstract/document/4814954/>

<https://www.mathworks.com/products/polyspace-code-prover.html>

https://vector.com/vi_canalyzer_en.html