Diplomarbeit

Entwicklung und Implementierung von OBD Funktionen für das AVL Steuergerät AVL RPEMS

Lukas Raschendorfer

Institut für technische Informatik
Technische Universität Graz
Vorstand: O. Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Römer

Betreuer: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Eugen Brenner

Graz, im November 2013

In Kooperation mit:

AVL List GmbH

Kurzfassung

Um die Entwicklung von neuen Motor- und Antriebskonzepten zu unterstützen hat die AVL List GmbH eine eigene Motor- bzw. Fahrzeugsteuerung entwickelt, die AVL RPEMS. Diese Steuerung wird in Motor- und Fahrzeugprototypen verwendet, um neue Funktionalitäten zu testen ohne dafür auf die Hilfe eines Industriepartners im Bereich Motorelektronik angewiesen zu sein.

Eine Funktion, die bisher noch im Funktionsumfang der RPEMS fehlte, war die Möglichkeit Diagnosetätigkeiten außerhalb der Entwicklungsumgebung durchzuführen – ein Feature, das besonders bei Systemen in Kundenhand wichtig ist.

Das Ziel dieser Arbeit ist es, eine solche Diagnosefunktion in das RPEMS Softwarepaket zu integrieren.

In einer Evaluierung werden verschieden Konzepte bezüglich Funktionalität, Kosten und Wartungsaufwand verglichen, wobei sich eine Implementierung des OBD II Diagnoseprotokolls als die praktikabelste Lösung herausstellt.

Die Standards, auf denen OBD II aufbaut werden dann untersucht, um herauszufinden, welche Teile für diese spezielle Aufgabe relevant sind. Danach wird die Architektur der bestehenden Software analysiert um die Interfaces zum Diagnosesystem festzulegen und ein Layout der Diagnosesoftware zu definieren, welches konsistent mit dem Rest des Systems ist.

Die fertige Implementierung wird erst mit üblichen Diagnosetools getestet und schließlich noch die Standardkonformität mit Hilfe von automatisierten Tests nachgewiesen.

Abstract

To assist the development of engine and drivetrain concepts, AVL List GmbH has developed its own Engine/Vehicle Control unit, the AVL RPEMS. This unit is used on prototype engines and vehicles to test new functionality without having to resort to an industry partner for the control system.

A function that was still missing from the RPEMS system was the ability to do diagnostics independent from the development environment – a feature especially important for systems in customer hands. This thesis' goal is to integrate such a system into the existing RPEMS software package.

In an evaluation of possible concepts to implement diagnostics, different solutions are compared in price, functional range, and support effort. From this, an implementation of the OBD II diagnostic protocol emerges as the most feasible solution.

The standards relating to OBD II are examined to determine which parts are relevant to this specific task. The architecture of the existing software system is then analyzed to determine the interfaces to the diagnostic system and to define a layout of the diagnostic module that is consistent with the rest of the system.

The finished implementation is then tested against common diagnostic tools as well as verified to be compliant with the standard using automated testing.

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am.. ....

(Unterschrift)

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....

date (signature)

Table of contents

List of Figures ..... **VI**

List of Tables ..... **VIII**

1 Introduction **1**

1.1 Motivation and Goal 1

1.2 Document Structure 2

2 Overview of Diagnostic systems **3**

2.1 History of OBD 3

2.2 Situation in Europe / EOBD 4

2.3 Future developments, WWH and UDS 5

2.4 OBD II standards 5

2.4.1 Applicable standards 5

2.4.2 OBD II Connector 7

2.4.3 CAN Communication 8

2.4.4 The generic tester 8

2.4.5 Manufacturer-specific extensions to OBD II 9

2.4.6 ISO 15765-2 Transport Protocol (ISO-TP) 9

2.4.7 The OBD II standard services in detail 11

2.5 The CAN Bus 20

3 Concepts **27**

3.1 Present scenario 27

3.2 External data logger 27

3.3 AVL specific software solution 28

3.4 OBD II Diagnostics Support 29

3.5 Rating matrix 30

4 Design **32**

4.1 Specification AVL RPEMS Future 32

4.2 Requirements for the OBD II implementation 32

4.3 Toolchain 33

4.4 Current Architecture of the RPEMS FUTURE System 35

4.4.1 Task scheduler....35

4.4.2 Drivers....36

4.4.3 Data Consistency....37

5 Implementation....40

5.1 Modifications to existing Software....40

5.1.1 CAN Software Stack....40

5.1.2 EEPROM Driver....42

5.2 New Software Modules....44

5.2.1 Important Shared OBD variables, constants and calibration tables....44

5.2.2 The OBD State Machine....44

5.2.3 The OBD Remapper....45

5.2.4 The OBD Persistent Storage Interface....47

5.2.5 The OBD Vehicle Information Interface....54

5.2.6 The OBD Debouncer....56

5.3 Diagnostics Implementation example....57

5.4 Architecture of the OBD subsystem....58

5.4.1 Timing....58

6 Testing, Results and Outlook....61

6.1 Test Setup....61

6.2 Early Testing....62

6.3 Communication tests....63

6.4 Systematic testing....65

6.5 Conclusion and Outlook....66

Bibliography....67

Appendix....70

Appendix 1: List of Abbreviations....71

Appendix 2: CANoe.DiVa Test Specification....72

Appendix 3: CANoe.DiVa Test Results....73

Appendix 4: Software Module Block Diagrams....74

Appendix 5: Source Code....83

List of Figures

Figure 2-1: OBD II standards and OSI Layers ... 6
Figure 2-2: OBD connector schematic (female, facing front) ... 7
Figure 2-3: OBD connector in vehicle ... 7
Figure 2-4: ISO-TP single and multiple frame communication ... 9
Figure 2-5: KWP2000 communication ... 12
Figure 2-6: KWP2000 application layer message ... 12
Figure 2-7: DTC Encoding ... 15
Figure 2-8: ISO 11898 in the OSI model ... 21
Figure 2-9: Schematic of a typical CAN transceiver ... 21
Figure 2-10: Typical CAN Bus signal levels ... 22
Figure 2-11: Composition of a CAN frame ... 23
Figure 2-12: CAN arbitration example ... 23
Figure 2-13: CAN error states ... 25
Figure 3-1: current diagnostic scenario ... 27
Figure 3-2: Data Logger System Overview ... 28
Figure 3-3: Proprietary diagnostic system overview ... 29
Figure 3-4: OBD II diagnostics system overview ... 29
Figure 4-1: AVL RPEMS toolchain ... 33
Figure 4-2: ASCET component hierarchy ... 34
Figure 4-3: RPEMS driver structure ... 36
Figure 4-4: Data becoming inconsistent as result of a race condition ... 37
Figure 4-5: Avoiding the race condition ... 38
Figure 4-6: RTA task priority scheme ... 38
Figure 4-7:cooperative scheduling example ... 39
Figure 5-1: Block diagram of the CAN stack ... 40
Figure 5-2: Block diagram of the modified CAN stack ... 42
Figure 5-3: EEPROM page structure ... 43
Figure 5-4: main OBD state flow diagram ... 45

Figure 5-5: usage of the ASCET variable model by the OBD remapper......46
Figure 5-6: OBD persistent storage memory organization......48
Figure 5-7: persistent storage element allocation logic......49
Figure 5-8: "Clear Memory" Program Flow......49
Figure 5-9: "Store DTC" Program Flow......50
Figure 5-10: "Read DTC" Program Flow......51
Figure 5-11: deferred DTC storing......51
Figure 5-12 : Program flow for storing and retrieving FF data......53
Figure 5-13: ISO-TP state flow......54
Figure 5-14: Debouncer behaviour......57
Figure 5-15: Exemplaric implementation of trottle pedal diagnostics......57
Figure 5-16: OBD software structure......58
Figure 5-17: OBD response time......60
Figure 6-1: Test Hardware Setup......61
Figure 6-2: INCA I Experiment showing data from the OBD remapper......63
Figure 6-3: CANalyzer used for communication tests......64
Figure 6-4: Successful DTC readout using the Autel scan tool......64
Figure 6-5:Torque App displaying data from OBD II services 02_H 03_H and 09_H......65

Picture sources are indicated in footnotes where appropriate.

Figures without reference were created by the author.

List of Tables

Table 2-1: OBD connector pinout.......... 7
Table 2-2: ISO-TP frame types.......... 10
Table 2-3: ISO-TP frame structure.......... 10
Table 2-4: Flow Control Flag values.......... 11
Table 2-5: Separation Time encoding.......... 11
Table 2-6: Physical and Functional Addressing.......... 11
Table 2-7: Service 1 Request.......... 13
Table 2-8: Service 1 Response.......... 13
Table 2-9: Service 1 Supported PIDs Response.......... 14
Table 2-10: Service 2 Request.......... 14
Table 2-11: Service 2 Response.......... 15
Table 2-12: Service 3 Request.......... 16
Table 2-13: Service 3 Response.......... 16
Table 2-14: Service 4 request.......... 16
Table 2-15: Service 4 response.......... 16
Table 2-16: Service 4 negative response.......... 17
Table 2-17: Service 6 request.......... 17
Table 2-18: Service 6 Response.......... 18
Table 2-19: Service 8 request.......... 19
Table 2-20: Service 8 Response.......... 19
Table 2-21: Service 9 Request.......... 19
Table 2-22: Service 9 Response.......... 19
Table 2-23: CAN Bus signal levels.......... 22
Table 2-24 : Bit stuffing example.......... 24
Table 3-1: Concept rating matrix.......... 31
Table 5-1: List of INFOTYPES.......... 55
Table 5-2: processes needed to answer an OBD request.......... 59
Table 6-1: Test Hardware.......... 61
Table 6-2: Test Software.......... 62

Chapter 1

Introduction

One of the business areas of AVL List GmbH (hereafter AVL) is the development of engines and engine control strategies for customers (mostly automotive OEMs). In the later phases of a development project the engine has to run on an engine dynamometer in order to advance mechanical and thermodynamic development. To actuate the ignition and injection systems of modern engines, an electronic control unit is required. In production, ECUs, from e.g. Bosch and Siemens, are used but this is not straightforward possible in the development phase. Many projects, especially ones with exotic or novel mechatronic concepts or innovative control strategies, require corresponding functions within the control system. These functions can only be implemented by the ECU vendor – a process associated with significant costs and lead time - and therefore not suitable for a system in development.
For this reason AVL in-house developed a bespoke control unit – the AVL RPEMS Future (Rapid Prototyping Engine Management System, hereafter RPEMS), a system where AVL has full control over both hard- and software and is thence able to implement new functions in a short timeframe. Because of the versatility this system provides, it can both be used as engine management and as VCU (Vehicle Control Unit) for vehicles with alternative powertrains (e.g. plug-in hybrids)
As a part of development projects, demonstrator vehicles are built on a regular basis and are used to calibrate and evaluate new engines and strategies in a real-world environment. These vehicles are not only used by the development engineers but often also by the customers and, especially with very innovative concepts, these vehicles get to be presented on trade shows and conferences.
It lies in the nature of a prototype that every now and then a problem appears that effectively disables the demonstrator vehicle. If there is no developer with the required specialized soft- and hardware present it is very difficult to correctly diagnose the problem and, if possible, get the vehicle working again.
For this reason AVL was looking for a possibility to perform simple diagnostics on a RPEMS that do not require the deployment of expensive calibration software and trained experts.

1.1 Motivation and Goal

As a mechanical engineer it may seem surprising for me to take on topic centered almost entirely on electronics and software. Indeed, an interest and knowledge in these topics is not very common among students of mechanical engineering, so my limited experience as

an electronics hobbyist predating my studies at TU Graz was enough to put me in the position of "Head of Electronics" on the TU Graz Racing Team[1]. There, in a steep learning curve, I came in contact with a number of automotive electronic components and control systems. This intensified my interest in the topics of electronics and (embedded) software development, so I began taking courses intended for students of telematics and also continued to work on applications of electronic systems in motorsport.

Working on the topic of diagnostics at AVL provided the opportunity to get a glimpse of the development and inner functioning of modern automotive ECUs, after I had been working with them as end user for some time.

The goal of this thesis was to develop a practical diagnostic solution on the AVL RPEMS ECU for use in the field while deepening my understanding of automotive embedded systems development.

The resulting solution should be easy to use, lightweight and robust and enable users to quickly diagnose problems with RPEMS units on-site. Ultimately, this should reduce downtime and support workload for the development engineers, because simple defects can be detected and solved without remote assistance.

1.2 Document Structure

This document describes the design and implementation of the OBD II diagnostic protocol on the AVL RPEMS in five chapters.

Chapter 2 depicts the historical and technical background of the OBD II protocol and gives an introduction to the fundamentals of CAN communication and real time systems.

The process leading up the decision to use the OBD II protocol to address the need for diagnostics on the RPEMS as well as competing solutions are described in chapter 3.

The existing software and hardware system and the design choices made on that basis are characterized in chapter 4.

Chapter 5 details the resulting implementation and presents the individual software components created for this project.

An exemplary usage scenario, test results and an outlook of possible future work can be found in chapter 6

[1] TU Graz Racing is a student club competing in the international Formula SAE Series and supported by TU Graz

Chapter 2

Overview of Diagnostic systems

This chapter gives an overview over current on-board diagnostic systems, focusing on OBD II, and the associated standards and regulations.

2.1 History of OBD

Contrary to popular opinion it is not European but US authorities who are leading in emission and OBD legislation. Special attention deserves California's CARB, which was and still is leading in the definition of many emission regulations, including OBD.

In 1940 California's population had already crossed the seven million mark. It is noteworthy that at the same time there were already 2.8 million vehicles registered which covered over 38 billion kilometers per year. [1]

Due to the special geographic and climatic conditions around the city of Los Angeles, it is especially vulnerable to the "SMOG" phenomenon (a portmanteau of "smoke" and "fog"), which is described as drastically increased concentration of air pollutants with simultaneously occurring decreased vision.

This phenomenon already led to first SMOG occurrences in 1943, which were rather serious with people suffering from smarting eyes, respiratory discomfort, nausea, and vomiting. [2] Also, visibility was less than a hundred meters.

Shortly thereafter, the "Bureau of Smoke Control" was founded and newer studies suggested that state-wide measures were needed.

In 1959 California's Department of Health set statewide quality standards for air quality including the concentration of sulfur dioxide, nitrogen dioxide, carbon monoxide and particulate matter.

1960 the "Federal Motor Vehicle Act of 1960" was enacted, which required federal research to address air pollution from motor vehicles. [3]

Already 1961 the first technological solution for emission reduction was presented, the positive crankcase ventilation (PCV). It made sure that blow-by gasses containing hydrocarbonate emissions would not be ventilated to the environment uncontrolled but would be withdrawn from the crankcase and returned to the combustion process together with fresh air and fuel.

1967 the "California Air Resources Board" merged multiple organizations and elaborated multiple, at this time, unique emission requirements for motor vehicles. [4]

After a slew of new regulations in the 70s and 80s had been successful in improving air quality, the focus was set on a new problem – maintaining the emission quality over a vehicle lifetime.

Until then, only the emission behavior of factory-new vehicles had been regulated, but due to modifications, maintenance, defects or wear it can quickly worsen. Monitoring of the emission behavior was deemed necessary. There are several possibilities to guarantee emission quality in the field:

- Periodical tests by the authority
- Self-diagnosing vehicles + control by the executive
- Self-diagnosing vehicles + control by the executive + periodical field tests

After pursuing the first solution with a mandatory bi-annual check in 1984 („CA SMOG Check Program“), finally the third solution became policy in 1988 when model year (MY) 1994 and later vehicles were required to have on-board computer systems that continuously monitor the emission performance of the vehicle and alert the driver if problems arose. Additionally, the authorities would do spot checks to ensure the systems were working as intended.

Integrated diagnostic systems were already required in the Smoke-Check directive from 1984 (OBD-I), which were introduced with the 1988 MY. These Systems should monitor emission-relevant systems, detect and store malfunctions and also activate a warning light if they do. The interface for accessing the fault memory was not standardized and was usually implemented by blink-codes.

The diagnostic system mandated from the MY 1994 (OBD II) required a correlation between the faults and the actual vehicle emissions. It should not only ensure the correct function of all involved systems but also monitor the chemical, mechanical, etc. functioning of all emission related systems by the means of “indirect” tests. An example for this would be monitoring the catalytic converter efficiency by means of an oxygen sensor both before and after the converter. [5]

If a fault is detected an OBD II compliant vehicle can not only store the fault itself but also the boundary conditions at the time of its occurrence.

Additionally, the OBD II standard defined the functional range and interface of a standardized tool (OBD Scan Tool) which is able to communicate with the vehicle’s fault memory and e.g. display the stored fault codes. Thanks to also standardized fault codes this also works across vehicle makes and models.

2.2 Situation in Europe / EOBD

Europe did not have legislation requiring standardized OBD up until the 2000s, however the introduction of increasingly complex microcomputer systems in vehicles necessitated diagnostic functions e.g. for end-of-line programming and in-shop diagnostics. Some of the technologies developed for this purpose (K-Line, KWP2000, and CAN) were then integrated into legislated OBD. Some vehicles, which were also sold on the North American market, (mainly from big manufacturers), were already OBD II compliant before this was required in Europe.

With the introduction of the EURO4 [6] emission regulations, OBD (in the form of EOBD) became mandatory for new vehicles in Europe starting with model year 2001 (gasoline engine) and 2004 (Diesel engine). The underlying technical standards, and thus the supported services and functional range are in all major regards identical to their OBD II counterparts. The remaining differences are mainly to be found in the mandated monitors. For example, the idle speed, fuel tank leakage and secondary air system only have to be monitored under US legislation. [7] With the ratification of newer legislation (Euro 6) the gap will be closed even further (see also 2.3). [8]

2.3 Future developments, WWH and UDS

In an effort to separate the diagnostic application from the transport protocol (e.g. KWP 2000, CAN/ISO-TP) the standard ISO 14229 - Unified Diagnostic Services (UDS) was created. It works partially similar to the OBD II diagnostic services, albeit with a much bigger functional range. [9] Nevertheless, it has not yet superseded the legacy protocols in OBD since legislative regulations still require their support.

In contrast to OBD, UDS is session-based. This means that the UDS server (meaning the ECU) has multiple sessions (states), which offer different services. While the default session has to just support session control, ECU reset and fault memory access, other sessions allow direct access to ECU memory and parameters, programming the ECU’s flash memory and even establishing an encrypted session for security related tasks.

Another effort currently underway would be the World Wide Harmonized (WWH) – OBD. It focuses on the separation of regulations regarding OBD from the specific emission regulations. The type of diagnostics performed and the diagnostic interface would be regulated by WWH-OBD while the emission thresholds would be defined by local legislation. The standard ISO 27145 has been created for this purpose. The technology used will be UDS over CAN and other interfaces (e.g. Ethernet), incorporating the existing definitions for fault codes and data items SAE J1979-DA and SAE J2012-DA to support a smooth transfer from existing standards. [5]

2.4 OBD II standards

2.4.1 Applicable standards

The function and functionality of OBD II is completely covered by several ISO standards, most of which are based on SAE standards. Since OBD II supports multiple physical media (K-Line, PWM, CAN) the standards describing them have to be considered as well. This thesis only covers diagnostics via CAN, so the relevant standards are:

- ISO 11898 – Controller Area Network
- ISO 15031 – Communication between Vehicle and external equipment for emissions-related diagnostics

- ISO 15765 – Diagnostics on Controller Area Network (CAN)
- SAE J1979-DA – Digital Annex of E/E Diagnostic Test Modes (PID definitions)
- SAE J2012-DA – Diagnostic Trouble Code Definitions

Figure 2-1 shows the relationship between the different standards and also their position in the OSI model.

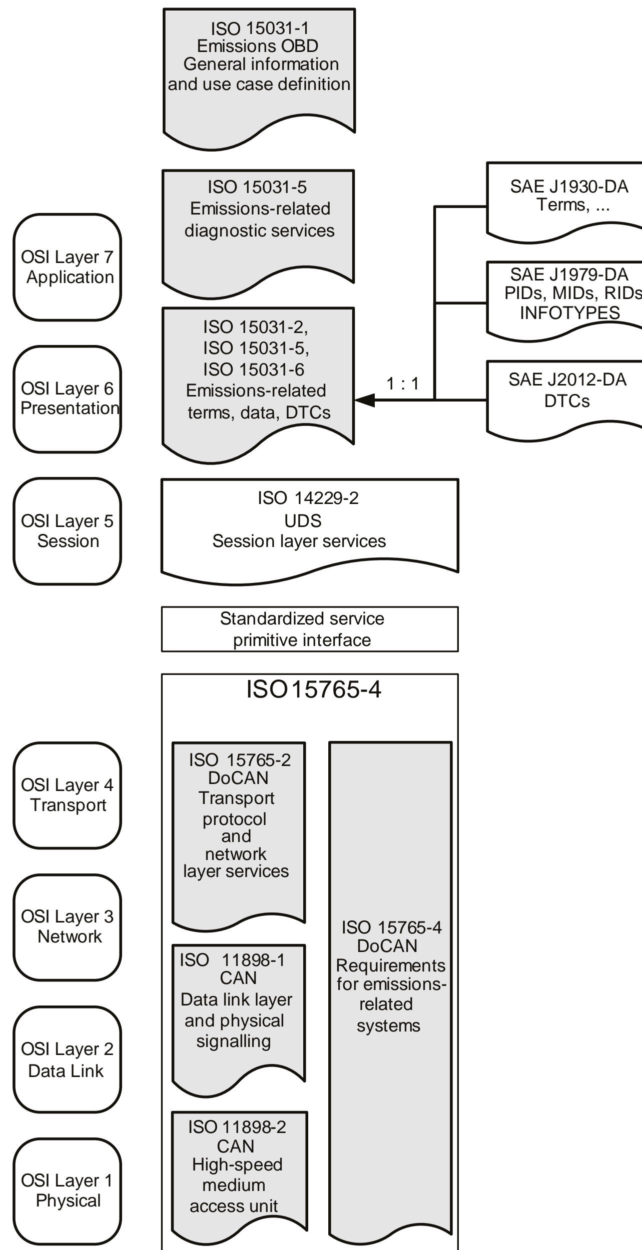


Figure 2-1: OBD II standards and OSI Layers[2]

[2] Modified from [22], Figure 2

2.4.2 OBD II Connector

ISO 15031-3 specifies the mechanical and electrical characteristics of a common, manufacturer-independent vehicle diagnostics connector with the intention of enabling the owner, garages and authorities to access any vehicle's diagnostic systems with a standardized device. Almost all commercially available testers feature the counterpart as their standard connector, sometimes with adapters for non-standard connectors (e.g. certain BMW and Mercedes models predating legislation that mandates the standard connector)

Table 2-1: OBD connector pinout

Pin	Function
1	Discretionary [3]
2	Bus positive line of SAE J1850 [4]
3	Discretionary[3]
4	Chassis ground
5	Signal ground
6	CAN_H line of ISO 15765-4 [4]
7	K line according to ISO 9141-2 and ISO 14230-4 [4]
8	Discretionary [3]
9	Discretionary [3]
10	Bus negative line of SAE J1850 b
11	Discretionary [3]
12	Discretionary [3]
13	Discretionary [3]
14	CAN_L line of ISO 15765-4 [4]
15	L line according to ISO 9141-2 and ISO 14230-4 [4]
16	Permanent positive voltage

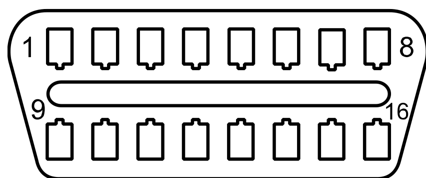


Figure 2-2: OBD connector schematic (female, facing front)[5]

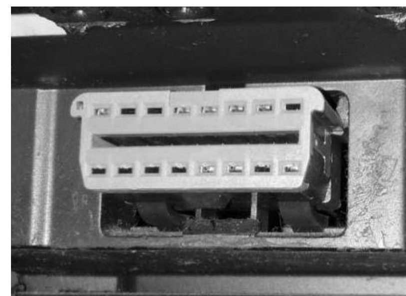


Figure 2-3: OBD connector in vehicle

[3] Assignment to this pin is left a t the discretion of the vehicle manufacturer
[4] This line may have an alternate assignment besides the diagnostic function
[5] Own artwork, based on [24]

Since pin 16 supplies 12 Volts from the car's battery, testers with moderate power requirements[6] supply themselves off the OBD connection and do not need a dedicated power source.

2.4.3 CAN Communication

As mentioned above, the implementation described in this thesis uses CAN as the physical medium for diagnostic information which is covered in ISO 15765. There the following properties are mandated for a diagnostic CAN interface:

- The standard baud rates for diagnostic services via CAN are 250 and 500kBit/s. [7]
- The external tester shall be an unterminated node on the bus.
- The CAN message address format can either be 11-bit (standard) or 29-bit (extended)[7]

2.4.4 The generic tester

Partly because of the confusion caused by the manufacturer-specific error readout methods during the OBD I era of emission regulations, OBD II regulations define a generic tester, which has to support all of the public (regulated) OBD services, as well as all the regulated communication protocols. This, on one hand, assures the user of such a device, that the tester will work on any (compliant) vehicle, that it will not damage the vehicle and that the readouts will be correct. On the other hand, it relieves the developers of the vehicle's diagnostic systems of the need to design and test their implementation against multiple, differently specified, systems. [10]
Since the OBD II implementation on the RPEMS should have the best possible compatibility, it is assumed that the connected tester does not exceed the functionality described in [10].
As mentioned above, the generic tester must support all the OBD II standard services – from the tester's perspective this means the following minimum functionality: [10]

- Continuously obtaining diagnostic trouble codes (DTCs) from the vehicle, displaying either its code, the related descriptive text (specified in SAE J2012-DA), or both to the user
- Displaying the current values of monitored data (from the data items defined in SAE J1979-DA) in the required format (e.g. value + unit)
- Displaying the data items in a freeze frame in the SAE J1979-DA specified format
- Results from monitors and tests as described in SAE J1939

[6] ISO15031-3 mandates that the 12V supply should be able to source at least 4A, meaning that the test equipment may draw at least 48W of power.
[7] The actual speed and address format is determined by the tester during the initialization sequence, only one combination needs to be supported

- Clearing the DTC, freeze frame and diagnostic test status data and having the user confirm this operation.
- Displaying OBD status information such as readiness tests and MIL status

2.4.5 Manufacturer-specific extensions to OBD II

Since the OBD II service is mandated on all new vehicles anyway, lots of OEMs have decided to extend the OBD II services with their own diagnostic services. This is usually done by defining non-standard PIDs for the standard services (Live Data, Freeze Frame), custom DTCs or even additional services like component tests. Some of these manufacturer-specific extensions could possibly add interesting features to the implementation but since they are only supported by very few and specialized testers and the specifications are hard to come by and/or very expensive their inclusion was deemed not useful.

2.4.6 ISO 15765-2 Transport Protocol (ISO-TP)

ISO 15765-2 covers the Network Layer (OSI Model Layer 3) services used in diagnostics via CAN. It describes a network layer protocol for data exchange between nodes on a CAN network and enables them transmit data in excess of the eight bytes offered by ISO 11898 CAN by segmenting the data if needed:

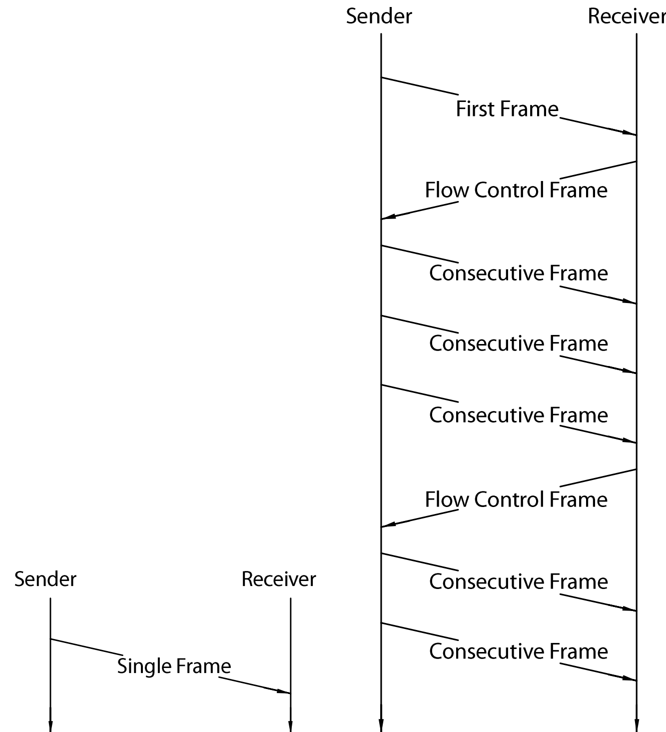


Figure 2-4: ISO-TP single and multiple frame communication[8]

[8] Taken from [25]

As apparent from Figure 2-4, multiple frame types exist. A “frame” in this context is a single CAN message, which encodes the frame type in its first byte.

Table 2-2: ISO-TP frame types

Frame Type	Type Code	Description
Single	0	This frame type contains the complete payload of the transmission (up to seven Bytes)
First	1	First frame of a multi-frame segmented transmission (more than seven bytes). Contains the complete payload size along with the first bytes of the data.
Consecutive	2	Contains subsequent data of a multi-frame transmission
Flow Control	3	Response of the receiver to a “First Frame”. It contains the parameters for the transmission of the subsequent frames.
<reserved>	4..15	<reserved>

Table 2-3: ISO-TP frame structure

Bit offset[9]	0..3		4..7	8..15	16..63
Single	0	Size	Data	Data	Data
First	1	Size			Data
Consecutive	2	Index		Data	Data
Flow	3	FC Flag	Block Size	Separation Time	Data

The “Data” fields in Table 2-3 denote actual “payload” data. Single and consecutive frames can carry up to seven bytes of data, first frames up to six.
The “Size” field encodes the number of used data bytes (0-7) in single frames, since they always have a DLC (size) of eight. The unused bytes are padded, usually with 00_H or 55_H.
Since the type code of the single frame is zero, the complete first byte is identical to <size> and the frame format can also be interpreted as a simple length-data type encoding.
In a first frame the “size” parameter is allowed to be as high as 4095, its full (unsigned) range, and denotes the net transmission size, meaning only payload data bytes. This tells the receiver how many frames it has to expect and how big a buffer it needs to allocate.
Consecutive frames encode an index field into the first byte which can take values between 0 and 15.
The index starts at zero and increments with each frame rolling over to 0 after 15. This allows the receiver to detect missing (dropped, lost) frames in a transmission. At the start of a transmission, the first frame is considered the 0th frame and the first consecutive frame will have the index 1.

[9] Motorola format, MSB first

The FC flag encodes a general response (Table 2-4), indicating if the receiver is at all (due to software or hardware limitations) able to accept a transmission of the size indicated in the First Frame.

Table 2-4: Flow Control Flag values

FC value	Name	Description
0	Clear to send	Indicates that the receiver is ready for reception
1	Wait	Indicates to the transmitter that the receiver is not yet ready for reception. The receiver then waits for another F/C frame.
2	Overflow/abort	Indicates that the receiver has to abort the transmission

The “Block Size” (BS) field tells the transmitter how many consecutive frames the receiver will accept before issuing another flow control frame. This allows receivers to split transmissions bigger than their receive buffer size into multiple parts. A value of zero means that all remaining frames can be sent at once.
The “separation time” (ST) (Table 2-5) tells the transmitter how long it has to wait between frames to allow the receiver to e.g. run its CAN stacks interrupt code.

Table 2-5: Separation Time encoding

ST	Description
0..127	separation time in milliseconds (ms)
$F1_H$..$F9_H$	100..900 microseconds (µs) in 100 µs steps

2.4.7 The OBD II standard services in detail

The services described in ISO 15031-4 are implemented in 10 “services” or “modes” with an assigned service ID (SID) ranging from 01_H to $0A_H$.
Generally speaking, the services all work in the same way: the tester sends a request and receives an answer encoded according to ISO 15765-2 from one or more ECUs.
To enable the tester to query either all of the available ECUs or a specific unit, the system of CAN Message IDs laid out in Table 2-6 is used.

Table 2-6: Physical and Functional Addressing

Sender	Receiver	CAN Msg. ID	Description
Tester	All ECUs	$07DF_H$	Functional Addressing – all ECUs listen to this ID
Tester	Specific ECU	$07E0_H$…$07E7_H$	Physical Addressing – Each ECU listens to a different ID to address it specifically
ECU	Tester	$07E8_H$…$07EF_H$	ECU Reply Address – This is the ECU’s physical address incremented by 08_H

The protocol used on the application layer is the KWP2000 (Key Word Protocol 2000) diagnostic protocol, which is designed so that all communication is initiated by the tester (Figure 2-5). The tester is therefore referred to as the "client" and the ECU as the "server". Communication is started by the tester's diagnostic application sending a message containing the diagnostic request to ECU over the network. The application layer informs the diagnostic application on the ECU of the request (indication). The ECU's response is then sent back over the network where the tester's application layer transmits the response to the tester application (confirm).

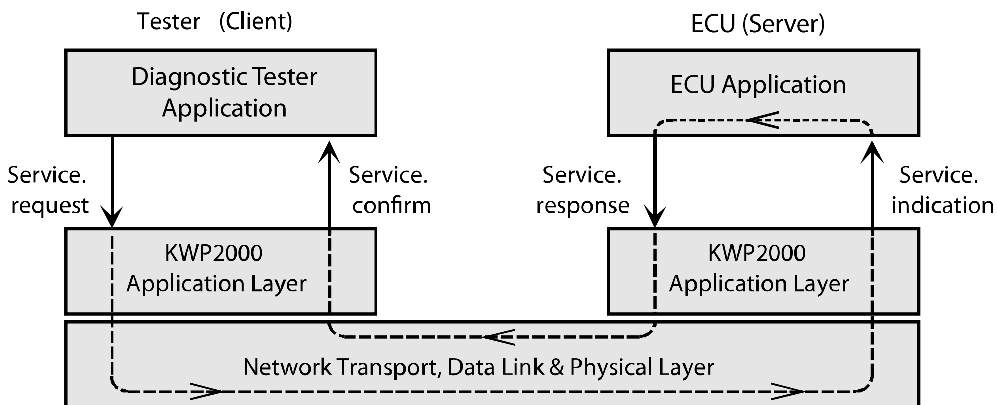


Figure 2-5: KWP2000 communication[10]

Services of the application layer consist of the following parts (Figure 2-6):

- Address information AI
- Service identifier SID
- Parameters, depending on the specific service.

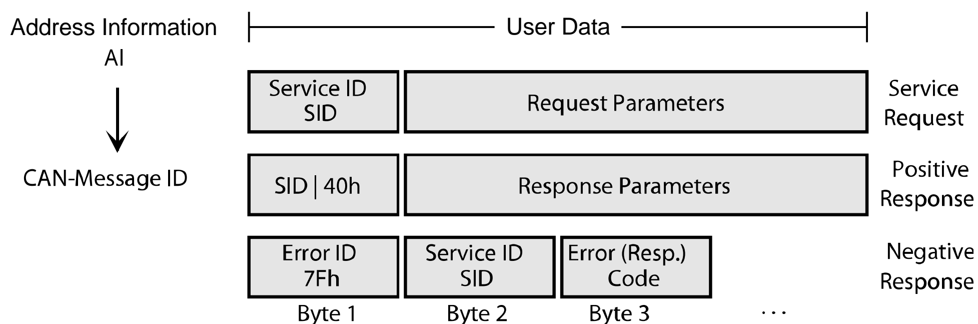


Figure 2-6: KWP2000 application layer message[10]

Address information is encoded in the message header (CAN ID), the service identifier (SID) is transmitted in the first byte of the user data[11], followed by the parameters. Depending whether the response is positive or negative, the response user data starts either with (SID + 40_H) or a standard defined error ID (see section Service 04_H below).

[10] Translated from [9]
[11] See ISO 15765-2 Transport Protocol (ISO-TP) (2.4.6), also referred to as "payload"

The following sections will detail the services offered by OBD II over CAN, with the first section, Service 01_H, covering many of the fundamentals.

Service 01_H

Service 1 provides "live" information. This can be current data from sensors or other sources, status information about the OBD system and basic capability information. All the available information is indexed by parameter IDs (PIDs), which are defined in SAE J1979-DA and referenced by ISO 15031-5.

Table 2-7: Service 1 Request

Byte#	Field	Example	Comment
1	SID	01_H	live data
2	PID	$0C_H$	engine speed

Table 2-7 shows the layout of a request frame. Since the request is only 2 bytes long (SID, PID), the single frame format is sufficient. The example request is for the engine speed (rpm), which has the PID $0C_H$ as defined in SAE J1939-DA.
The reply frame shown in Table 2-8 has a similar structure, mirroring the SID and PID from the request. This theoretically allows a (simpler and more robust) stateless software design in the tester, as request and response do not need to be correlated. Considering ISO 15765-4 defines that ECUs shall not respond to unsupported requests (i.e. non-implemented services, unsupported PIDs), this also makes the protocol more robust against timing glitches (e.g. responses arriving late).

Table 2-8: Service 1 Response

Byte#	Field	example	comment
1	SID + 40_H	41_H	live data reply
2	PID	$0C_H$	engine speed
3	PID Data	35_H	3450 rpm
4		$E8_H$	

Since the engine speed is encoded in a 16-bit variable, the total transmission (PID, SID, data) is just 4 Bytes and can be transmitted in a single frame. The size and encoding differs depending on the PID and can, for later additions to the standard (PID > 65_H) exceed the capacity of a single frame. SAE J1939-DA defines the encoding of PID $0C_H$ as $speed = \big((A \cdot 256) + B\big)/4$, effectively meaning that the value is encoded as unsigned 16-bit integer with a quantization of 0.25 rpm/bit. In this case it would be $35E8_H/4$ = $13800_D/4$ = 3450 rpm.
To determine which PIDs a certain ECU supports, the PIDs 00_H, 20_H, 40_H… each return a 32-bit ($32_D = 20_H$) bit field, which encodes the support for the 32 PIDs following them. So

PID 00_H contains this information for PID 01_H to 20_H, PID 20_H for PID 21_H to 40_H and so on. Usually the tester will query 00_H first, then, if supported, 20_H and so forth. PID 02_H is not supported in service 1.

Table 2-9: Service 1 Supported PIDs Response

Byte#	Field	example	comment
1	SID + 40_H	41_H	live data reply
2	PID	00_H	Supported PIDs in range $01_H..20_H$
3	PID Data	$2C_H$	2C10 0000_H =
4		10_H	1010 1100 0001 0000 0000 0000 0000 0000_B
5		00_H	meaning that
6		00_H	PIDs 01_H, 03_H, 05_H, 06_H, $0C_H$ are supported

Since, for most services, the tester is required to “know” the supported PIDs it will send a service 01_H request for PID 00_H using functional addressing. All ECUs supporting service 01_H will respond telling the tester not only the supported PIDs but also capacitating it to enumerate the available ECUs.

Service 02_H

Service 02_H is mostly identical to service 01_H, with the difference that the data included is not “live” but a snapshot (“freeze frame”) done at a certain point in the past. This point is usually when a malfunction is detected and an error (“DTC”) is stored in the fault memory. Service 02_H will also provide the error code of the fault that caused the storing of the freeze frame (PID 02_H, which is thus not supported in service 1).

OBD II implementations are only required to store one freeze frame, but it is allowed to store and retrieve up to 255. Since it is not required, not all testers support the viewing of additional freeze frames.

Since the ECU may not take a snapshot of all PIDs supported in service 01_H, it is required that the tester queries at least PID 00_H to determine the available PIDs in the Freeze Frame.

The layout of the service 02_H request and response frame is almost identical to service 01_H, except that in addition to PID and SID a Freeze Frame number (FFID) has to be supplied.

Table 2-10: Service 2 Request

Byte#	Field	example	comment
1	SID	02_H	Freeze Frame Data
2	PID	$0C_H$	engine speed
3	FFID	00_H	Freeze Frame #0

Table 2-11: Service 2 Response

Byte#	Field	example	comment
1	SID + 40_H	42_H	Freeze Frame Reply
2	PID	$0C_H$	engine speed
3	FFID	00_H	Freeze Frame #0
4	PID Data	35_H	3450 rpm
5		$E8_H$	

The valid range for FFID, which corresponds to the number of available Freeze Frames minus one, cannot be queried. However, if PID 02_H (responsible DTC) is queried and the reported value is 0000_H the data reported for this FFID is not valid (no Freeze Frame with this ID is available).

Service 03_H

Service 03_H gives access to stored fault codes (diagnostic trouble codes, DTCs). Their number can be determined by querying service 01_H, PID 01_H - a maximum of 127 is supported. The fault codes have a standardized format and meaning, given in SAE J2012-DA and ISO 15031-6. SAE J2012-DA also specifies certain fault code ranges as "manufacturer specific"; codes from this range can only be decoded with the knowledge of vehicle make & model and the manufacturer's code tables.

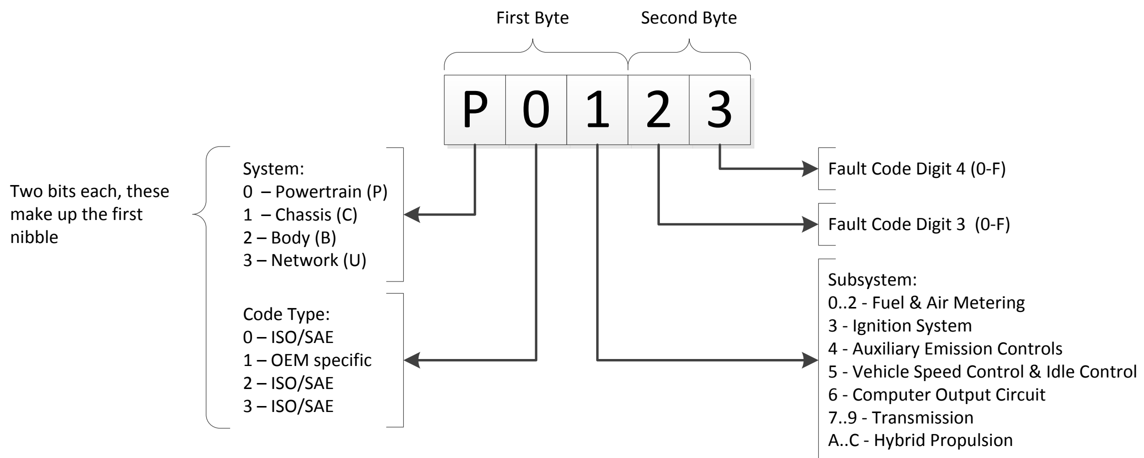


Figure 2-7: DTC Encoding

DTCs can only be received "in bulk"; access to individual codes is not possible. Since each DTC is two bytes in size, a maximum of two DTCs can be encoded in an ISO 15765-2 single frame transmission. Therefore service 03_H replies are usually segmented transmissions.

Table 2-12: Service 3 Request

Byte#	Field	example	comment
1	SID	03_H	Stored DTCs

Table 2-13: Service 3 Response

Byte#	Field	example	comment
1	SID + 40_H	43_H	Stored DTCs Reply
2	numDTC	$0E_H$	number of stored DTCs (14)
3	DTC 1	01_H	P0123
4		23_H	
...	...	...	...
28	DTC14	03_H	P0328
29		28_H	

Service 04_H

Service 04_H is used to clear the ECU's fault memory. This includes:

- MIL status
- DTCs and number of stored DTCs
- Pending DTCs
- Freeze frame data
- Sensor test data
- Status and results of tests and monitors
- Time and distance counters

Service 04_H only provides the possibility for a complete wipe of the fault memory, deleting specific DTCs or Freeze Frames is not possible. The ECU will respond with a negative response message if the memory cannot be cleared for whatever reason.

Table 2-14: Service 4 request

Byte#	Field	Example	Comment
1	SID	04_H	Clear fault memory

Table 2-15: Service 4 response

Byte#	Field	Example	Comment
1	SID + 40_H	44_H	Clear fault memory response

A negative response frame starts with the NR field, the Negative Response identifier, which is always $7F_H$. It is followed by the SID of the service that provoked the negative response and a Negative Response Code (NRCode). When using ISO 15035 CAN as diagnostic connection the only three allowed negative response codes are:

- 21_H – Busy Repeat Request – The ECU cannot process the request due to load
- 22_H – Conditions not correct – The requested data is not available under the current circumstances
- 78_H – Request Received Response Pending – The request is supported but the data is not immediately available.

Table 2-16: Service 4 negative response

Byte#	Field	Example	Comment
1	NR	$7F_H$	Negative Response
2	SID	04_H	Clear Fault Memory
3	NRCode	22_H	Conditions not Correct

A response code of 22_H or 21_H aborts the transmission, while 78_H means that the response will be delayed by a maximum of 5000ms. The delay can be extended by re-sending the negative response frame.

Service 05_H

Service 05_H provides oxygen sensor test data and is unavailable on implementations which use the CAN protocol. In these cases it is superseded by service 06_H.

Service 06_H

Service 6 provides test data from systems that are not /cannot be continuously monitored, like the catalytic converter.

The request which monitor IDs are supported by a system is similar to the corresponding request in service 01_H. When requesting supported monitor IDs, up to six ranges can be requested at once, while only one monitor ID is allowed in a regular request.

Table 2-17: Service 6 request

Byte#	Field	Example	Comment
1	SID	06_H	Monitor Result
2	OBDMID	$3D_H$	Purge Flow Monitor

The reply will include one or more monitor test results, depending on which tests are supported for a given monitor. Each test record contains the monitor and test ID (OBDMID, S/MDTID) followed by Information about the scaling and unit of the test value as per SAE J1979-DA. The last six bytes are composed from the actual test result value, and the associated upper and lower limit, each encoded in 16-bit values. Since the amount of data

requires a segmented transmission, the total number of tests can be derived from the message length parameter in the first frame of the transmission. The transmission of the actual value of a test result along with the limits instead of a pass/fail indicator is necessary since the pass/fail decision depends on all of the tests (one or more might be required/sufficient). If a test has not been completed at the time of the request, the fields MINTL, MAXTL and TV read zero.

Table 2-18: Service 6 Response

<table>
<tr><th>Byte</th><th colspan="2">Field</th><th>Example</th><th>Comment</th></tr>
<tr><td>1</td><td colspan="2">SID + 40_H</td><td>46_H</td><td>Monitor result reply</td></tr>
<tr><td>2</td><td rowspan="9">OBDMIDREC[12]</td><td>OBDMID</td><td>$3D_H$</td><td>Purge flow monitor</td></tr>
<tr><td>3</td><td>TID</td><td>01_H</td><td>Test ID</td></tr>
<tr><td>4</td><td>UASID</td><td>$0A_H$</td><td>Unit and scaling ID (0.122mV/bit)</td></tr>
<tr><td>5</td><td rowspan="2">TV</td><td>79_H</td><td rowspan="2">Test value (7934_H = 3.785V)</td></tr>
<tr><td>6</td><td>34_H</td></tr>
<tr><td>7</td><td rowspan="2">MINTL</td><td>59_H</td><td rowspan="2">min. test limit ($59A6_H$ = 2.800V)</td></tr>
<tr><td>8</td><td>$A6_H$</td></tr>
<tr><td>9</td><td rowspan="2">MAXTL</td><td>89</td><td rowspan="2">max. test limit ($89AE_H$ = 4.300V)</td></tr>
<tr><td>10</td><td>AE</td></tr>
</table>

Service 07_H

Service 07_H displays “pending” faults. Emission legislation allows certain faults to be “debounced” or “confirmed”, which means, that they may occur multiple times under certain circumstances before they are confirmed and an entry is generated in the vehicle’s fault memory (and, depending on the fault, the MIL is lit). Faults that have occurred but not yet passed this threshold are accessible via this service.
The communication is identical to service 03_H, except that a SID of 07_H is used.

Service 08_H

Service 08_H allows the connected tester to control certain on-board systems or tests, for example actuator tests. As of 10/2011 only two tests were defined in SAE J1979-DA: the “Evaporative System Leak Test” (not mandated in Europe) and a “Diesel Particulate Filter Regeneration Request”.
The request which test IDs are supported by a system is similar to the corresponding request in service 01_H.
Both the test request and response frames consist of the SID and the test ID (TID) and can contain up to 5 bytes of additional data (e.g. for configuring the test).

[12] A service 6 response may contain more than one OBDMIDREC, depending on how many tests (TIDs) are associated with the OBDMID.

Table 2-19: Service 8 request

Byte	Field	Example	Comment
1	SID	08_H	Request device control
2	TID	02_H	Particulate filter regeneration

Table 2-20: Service 8 Response

Byte	Field	Example	Comment
1	SID + 40_H	48_H	Request device control reply
2	TID	02_H	Particulate filter regeneration

Service 09_H

Service 9 provides general vehicle information (referred to as "infotypes"), like the VIN (Vehicle Identification Number), Calibration ID and ECU name.

The request which infotype IDs are supported by a system is similar to the corresponding request in service 01_H.

The vehicle information request frame just contains two data bytes: the SID and the infotype ID (INFTYP). The response message may contain more than one instance of the requested infotype (e.g. multiple VINs), therefore the third byte contains the number of infotype instances in the transmission. Since each infotype has a fixed size, the infotype data is padded with 00_H filler bytes.

Table 2-21: Service 9 Request

Byte	Field	Example	Comment
1	SID	09_H	Request vehicle information
2	INFTYP	$0A_H$	ECU name

Table 2-22: Service 9 Response

Byte	Field	Example	Comment
1	SID + 40_H	49_H	Vehicle information reply
2	INFTYP	$0A_H$	ECU Name
3	NODI	01_H	Number of Data Items (1)
4	data 1	45_H	'E'
5	data 2	43_H	'C'
6	data 3	$4D_H$	'M'
7	data 4	31_H	'1'
8	data 5	$2D_H$	'-'
9	data 6	41_H	'A'
10	data 7	56_H	'V'

Byte	Field	Example	Comment
11	data 8	$4C_H$	'L'
12	data 9	20_H	' '
13	data 10	52_H	'R'
14	data 11	50_H	'P'
15	data 12	45_H	'E'
16	data 13	$4D_H$	'M'
17	data 14	53_H	'S'
18	data 15	00_H	Filler bytes
19	data 16	00_H	
20	data 17	00_H	
21	data 18	00_H	
22	data 19	00_H	
23	data 20	00_H	

Service $0A_H$

Service 10 stores "permanent" DTCs. A DTC usually gets the "permanent" status when it is confirmed (debounced) and requires the MIL to be lit. It cannot be deleted by service 04_H but is automatically removed when the underlying fault is not detected any more (the removal is debounced, too)

The communication is identical to service 03_H, except for the SID of $0A_H$.

2.5 The CAN Bus

Ever since the introduction of microprocessor-controlled systems in vehicles it was necessary to exchange data between these systems. While the first solutions to this problem consisted mainly of simple switched lines or analog signals, more complex demands for off-board diagnostics (e.g. in the workshop) had OEMs looking for manufacturer-independent solutions for digital data communication.

BOSCH, as one of the leading European manufacturers of electronic control units at the time, introduced a simple (ISO9141, similar to RS232C/V.24) serial interface.

As the demands on in-car-interconnects rose, BOSCH, together with European OEMs introduced the CAN Bus, which was later standardized as ISO 11898.

The defining properties of CAN as a bus system are:

- Differential signaling
- Speed up to 1 Mbit/s
- Message-oriented event-triggered protocol
- Linear topology
- Multimaster architecture using CSMA/CR

Looking at the OSI Model (ISO 7498-1) for communication systems, ISO 11898 defines layer 1 (Physical Layer) and layer 2 (Data Link Layer). Layers 3-7 are not defined in ISO 11898 and are dependent on the specific application.

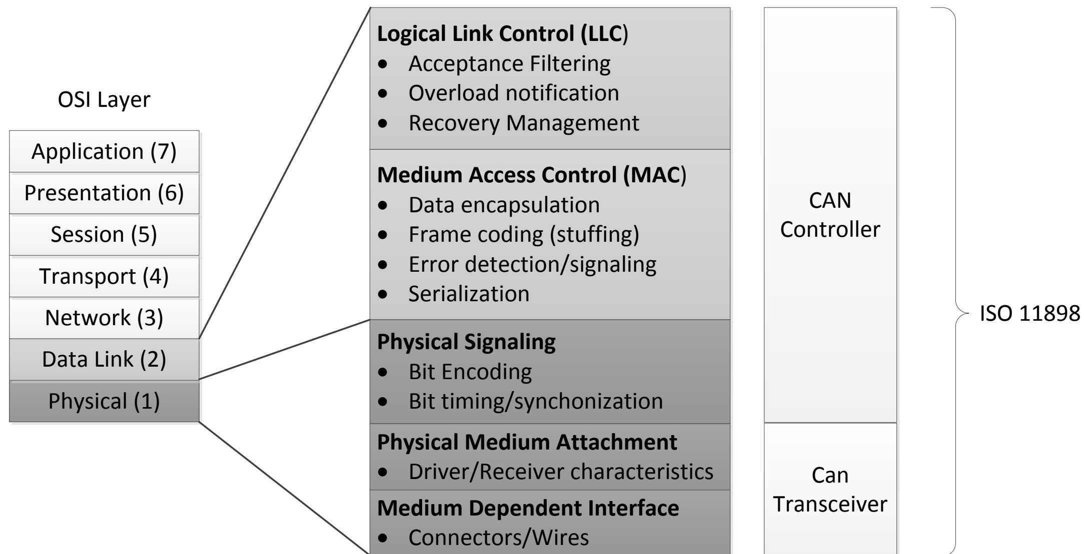


Figure 2-8: ISO 11898 in the OSI model[13]

The physical medium for CAN is usually a single twisted pair of wire, optionally shielded, one wire being the “CAN High” and the other one the “CAN Low” line. Both shielded and unshielded twisted pair lines are widely used and portray no significant cost impact in CAN Applications.

Data is transmitted by means of differential signaling, meaning the data is encoded in the voltage difference between the two wires. This gives the CAN bus excellent immunity against EMI since interference usually shifts the voltage levels in both conductors of twisted pair wiring (common mode interference) and leaves the differential voltage unaffected.

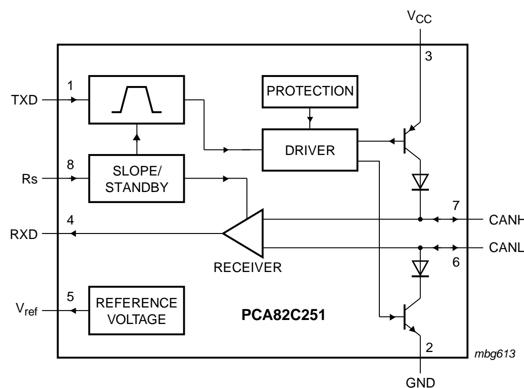


Figure 2-9: Schematic of a typical CAN transceiver[14]

[13] Redrawn according to [26],[27]
[14] Taken from [28]

Figure 2-9 shows the block diagram of a typical High-Speed CAN Transceiver, the NXP/Philips PCA82C251. It is clearly visible how both transmitter and receiver operate on the same lines, forcing half-duplex operation. This means that, while any node can both receive and transmit, it can only do one at a time. The RXD and TXD lines are the digital serial interface to the microcontroller and usually operate on TTL voltage levels.
The differential side of the CAN Transceiver has two defined states – dominant and recessive. These states are defined by the voltage levels shown in Table 2-23 and Figure 2-10 below.

Table 2-23: CAN Bus signal levels

	recessive			Dominant		
	min.	typ.	max.	min.	typ.	max.
V_{CANH}[15]		*2,5V*	*7,0V*		*3,5V*	*7,0V*
V_{CANL}[15]	*-2,0V*	*2,5V*		*-2,0V*	*1,5V*	
V_{Diff}	*-0,12V*	*0V*	*0,012V*	*1,2V*	*2,0V*	*3,0V*

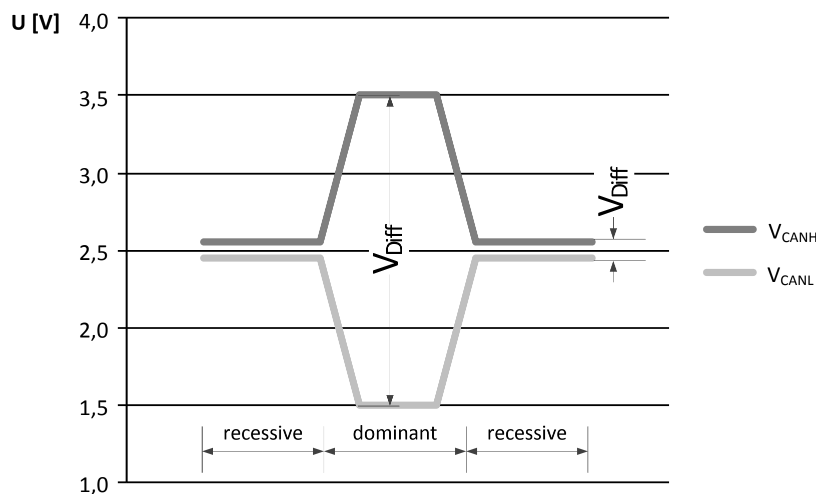


Figure 2-10: Typical CAN Bus signal levels

The recessive state, as the name suggests, is the state of the bus when all transmitters are set passive. Since the transmitter can only push or pull each line in one "direction" (CANL towards GND, CANH towards V_{CC}), a correctly wired CAN network cannot have transmitters working against each other, leading to possible damage. Also, monitoring the lines during a transmission allows the transceiver to detect collisions when the line is in its dominant state despite the transceivers own transmitter being passive.
Transmissions on the CAN bus are datagrams by the name of "frames"; a typical CAN frame is shown below. Note, that logic one is the recessive state, while logic zero is the dominant state. Also, the number of bits represents the net length. The physical transmission may contain additional stuff bits (see Table 2-24).

[15] V_{CANH}, V_{CANL} are common-mode voltages, i.e. measured towards the ground of the respective can node.

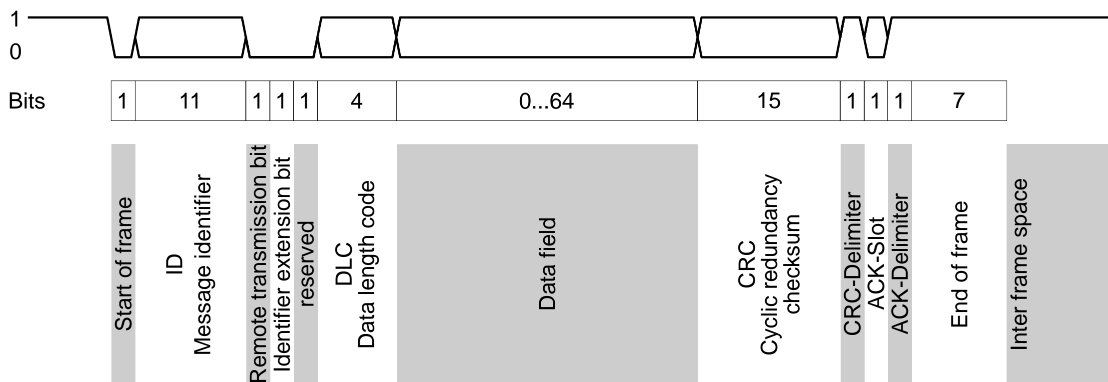


Figure 2-11: Composition of a CAN frame[16]

A transmitter willing to transmit a CAN frame first has to wait for the bus to become recessive for the „Inter Frame Space" (IFS) time, which is equivalent to three bit times[17]. Then it pulls the bus into the dominant state, indicating "Start of Frame" (SOF).

This is followed by eleven bits of message identifier (ID); during this time period the collision resolution is performed: If the transmitting node detects a "zero" (dominant state) when transmitting a "one" (recessive state) it stops the transmission by switching to the recessive state and waits for the next inter frame space to retry. This process is called "arbitration" and integrates a collision resolution with a prioritization mechanism (Carrier Sense Multiple Access / Collision Resolution – CSMA/CR). Since the logical zero is dominant and "wins" the arbitration, CAN frames with lower ID therefore have higher priority on the bus system meaning a higher chance of getting transmitted as the next frame.

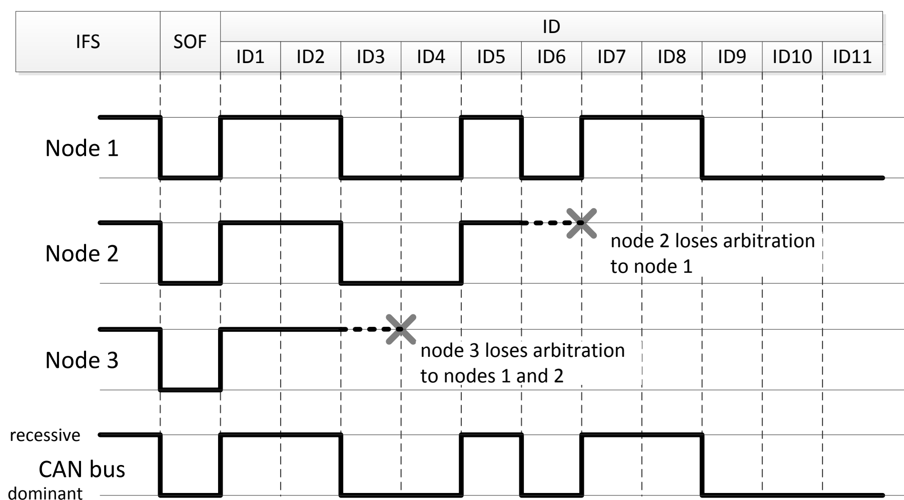


Figure 2-12: CAN arbitration example

The next bit after the ID is the "Remote Transmission Bit" (RTR) which indicates that the frame is a "Remote Request Frame". If an extended identifier is used, it becomes the SRR bit, which is also written dominant and is considered during arbitration.

16 Modified version of [29]

17 "bit time" means the time it takes to transmit a single bit and is calculated as 1/<bit rate >

The next bit is the identifier extension bit (IDE); if it is logic one, it indicates that this CAN frame uses a 29-bit identifier (extended) instead of an eleven-bit (standard) one. If this would be the case, the IDE Bit is followed by 18 more identifier bits, a replacement RTR bit and two reserved (dominant) bits. The IDE bit is considered during arbitration. When an eleven-bit identifier is used, the IDE bit is just followed by one reserved (dominant) bit.
The next four bits, which together with RTR and IDE form the “Control Field”, indicate the length of the following data section in bytes (DLC). Valid values are 0…8 (Bytes).
The data field consists of the “payload” data (DATA), the actual data that is transmitted by the CAN frame, and is variable in length, depending on the contents of the DLC field.
The data field is followed by 15 bits of CRC data which uses all the data from SOF to DATA as input.
A “CRC Delimiter” bit is followed by the acknowledge slot/bit (ACK). Any device on the bus that correctly received the frame up to this point will pull the bus in the dominant state and give the transmitter the information that the frame has been received by at least one node.
After the ACK bit and the “ACK Delimiter Bit” seven recessive bits will follow, those indicate the end of the frame (EOF). After the EOF and an IFS the bus nodes may send another SOF, starting the next frame.
To make sure that an ID or data sequence consisting of seven consecutive recessive bits falsely triggers EOF detection, a mechanism called “bit stuffing” is applied to all fields except ACK, EOF and the CRC delimiter.
It works by introducing (“stuffing”) an extra bit of inverse polarity after the fifth bit of a bitstream containing more than five bits of the same polarity, regardless of the polarity of the next bit in the stream. This can sometimes lead to additional stuffing to be necessary, as demonstrated in example 2 below.

Table 2-24 : Bit stuffing example

	Example 1	Example 2
Bitstream	01011111010	10100000111101
Stuffed bitstream	010111110010	1010000011111001
Note: 0 = dominant, 1= recessive. Stuff bits are underlined		

Besides the standard CAN frame, three “special” frames are specified in the CAN standard:

- Remote Request Frame
- Error Frame
- Overload Frame

A “Remote Request Frame” is a standard frame, in which the RTR bit is set (0) and the “Data Field” is missing. It indicates to (requests from) another CAN node in the network to send a standard frame with the indicated ID.

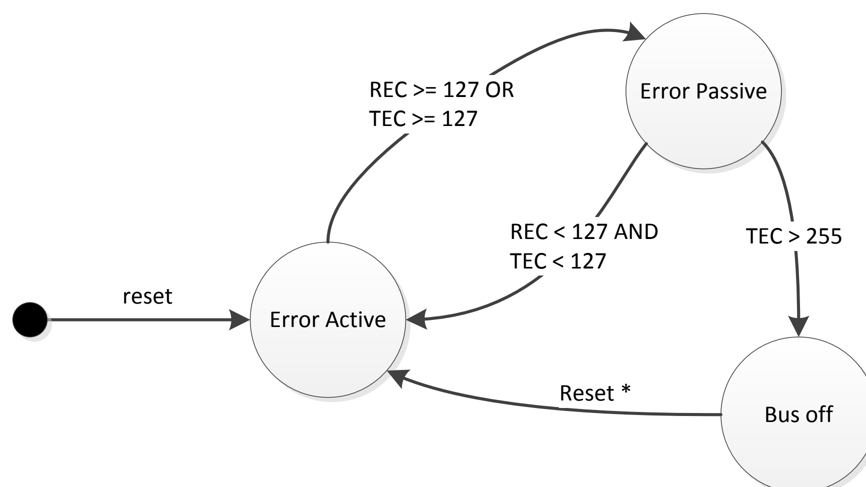
The error frame is used to notify other nodes in the network that a transmission error has occurred. There are multiple mechanisms to detect errors, e.g. incorrect bit stuffing and incorrect CRC. If an active network node detects an error, it pulls the bus to the dominant state for six bit times, starting with the next bit (active error frame). This is detected by the other nodes on the network which respond themselves by doing the same. Afterwards the bus is left at the recessive state for eight bits (error frame delimiter) and the communication restarts with an IFS. Error passive nodes (see below) may only send passive error frames, which start with six recessive bits so that they can mark their own transmission as erroneous but cannot disturb the transmission of other nodes. This concept assures that defects causing transmission errors are confined to the affected node so that it does not bring down the whole bus.

The overload frame is identical to the (active) error frame; with the exception that the start of the overload frame is fixed after the last bit of the EOF (i.e. it replaces/overwrites the IFS)

To prevent a faulty node from blocking the bus by continuously transmitting active error frames, each node has to implement an error handling and mitigation system.

It is implemented as a state machine consisting of three states, visualized in Figure 2-13:

- Error active (initial state, TX and RX active)
- Error passive (TX and RX active, must not transmit active error frames)
- Bus Off (disconnected from bus)



*) some implementations have an automatic transition back to this state as specified in the original BOSCH standard. The ISO implementation does not allow this.

Figure 2-13: CAN error states

The transition between these states is controlled by the transmit (TEC) and receive (REC) error counters, which are incremented or decremented according to the following rules:

- Frame received correctly — decrement REC by 1
- Receive fault — increment REC by 1
- Bit error while sending Error Flag — increment REC by 8
- Frame sent correctly — decrement TEC by 1
- Transmit fault — increment TEC by 8

Chapter 3

Concepts

As stated in chapter 1, the goal of this thesis is to integrate diagnostic capabilities into the RPEMS' software package. Naturally, diagnostics require not only an ECU that offers the corresponding capabilities but also an external unit that performs the diagnostics, what makes it part of the overall diagnostics concept.
Starting with the scenario currently in use, three additional concepts have been developed and compared on the basis of per-unit deployment cost, maintenance effort and ease-of-use for the person performing diagnostics.

3.1 Present scenario

Diagnostics are generally performed by monitoring ECU-internal values using the PC-based software ETAS INCA and a CAN CCP connection.
To interpret the readings, detailed knowledge of the software modules used on the ECU is needed. Fault events can only be registered / recorded while the PC is connected. Each additional unit that should have diagnostic abilities requires significant investment in Hardware and Software Licenses.

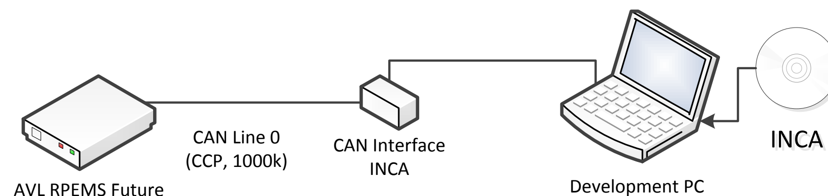


Figure 3-1: current diagnostic scenario

Hardware Cost (per unit): high (ETAS CAN card)
Software Cost (per unit): high (ETAS INCA license)
Development effort: none
Maintenance effort: none
Ease of use: low (expert knowledge required for both tools and data analysis)

3.2 External data logger

Each RPEMS could be paired with an external data logger which records CAN data provided by the ECU. A PC could then be used to download the data from either the logger or the storage medium it uses (e.g. SD Card) and analyze the data.
The modifications to the RPEMS software would at least include writing the variables and values intended for logging on the CAN bus. To be human-readable, in most cases a conversion would need to be performed.

As an example of this class of device the Kvaser Memorator HS/HS and the Vector GL1000. Both devices are stand-alone loggers need nothing more than a CAN connection and power. Upon startup, they capture the CAN traffic and save it to an SD Card. Both feature a ring buffer for continuous long-time recording and configurable event-triggered recording. Aside from their respective native formats, both loggers (and most competing products) support standard formats such as CSV[18]. [11], [12] This allows the data to be read by any PC with a card reading device and spreadsheet software such as Excel (or even a text editor). The interpretation of the data however, is still up to the user. Depending on the capabilities of the logger, the data will already be the individual data items in the correct format or only the raw CAN data, necessitating documentation for conversion.

Figure 3-2: Data Logger System Overview

The hardware cost per unit would be roughly similar to the cost of the ETAS hardware required for INCA – about $1000. However, no INCA license is needed, instead readily available software can be used, such as a text editor or excel. Another possibility would be less expensive generic data analysis software like GEMS GDA pro ($200).

Hardware Cost (per unit): high (CAN logger)
Software Cost (per unit): low (analysis software) to none
Development effort: low (adapt CAN software)
Maintenance effort: low (maintain conversion data or logger configuration)
Ease of use: medium to low (expert knowledge required for data analysis)

3.3 AVL specific software solution

To-be-designed PC Software could replace the basic functionality of ETAS INCA (display and recording of measurement data) and could also download stored faults from the ECU. This would waive the need for expensive ETAS hardware and licenses for the diagnostics PC.

Since PC software development cannot be done by the RPEMS embedded software developers, it would have to be outsourced to another unit in the company or external contractor. To estimate the cost for this software solution, it is assumed that the project will contain around 10.000 lines of code (LoC), a typical number for small applications. According to [13], the development pace for this type of software can be estimated at roughly 2500 LoC/staff month, leading to a 4 man-month development phase for the initial version

[18] Comma Separated Values, a format for storing tabular data in plain-text form

translating to over $10000 in labor cost. If the Software ought to contain “expert knowledge”, e.g. allowed values and pass/fail thresholds, it has to be continuously adapted to new functionality on the RPEMS, what considering the many and rapidly evolving ECU programs would require significant effort and expenditure.
The hardware cost would be a bit lower than in the current scenario, since many available CAN interfaces are much cheaper than the ETAS interface.
On the ECU side, the amount of changes depends on the architecture of the PC software. If it were to use CCP, only the fault memory would have to be implemented. If the PC software should communicate by a custom protocol, this would have to be implemented, too.

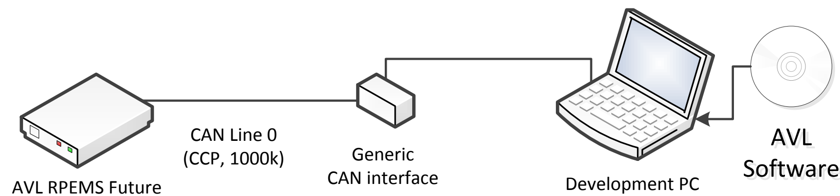


Figure 3-3: Proprietary diagnostic system overview

Hardware Cost (per unit): medium (generic CAN interface)
Software Cost (per unit): medium (high development cost spread over few units)
Development effort: high (implement fault storage and custom protocol)
Maintenance effort: high (maintain ECU and PC side of system)
Ease of use: high (software can be made as end-user friendly as necessary)

3.4 OBD II Diagnostics Support

The OBD II Diagnostic Protocol is implemented as a software module on the RPEMS. Stored faults and live measurements can be downloaded/displayed using an OBD II/EOBD compatible tester.
While the initial effort for creating the required ECU software modules is relatively high, it is only a one-time effort. All the maintenance work (adding new error codes etc.) can be done by the ECU software developers; since both the communication and the content is standardized, only one “end” has to be maintained. Compatible devices for data display are commercially available, as well as software testers for many platforms, including mobile devices, at a very low cost (starting at about $50). Alternatively, most internet-connected devices are able to gain diagnostic abilities in a matter of minutes by means of free software.

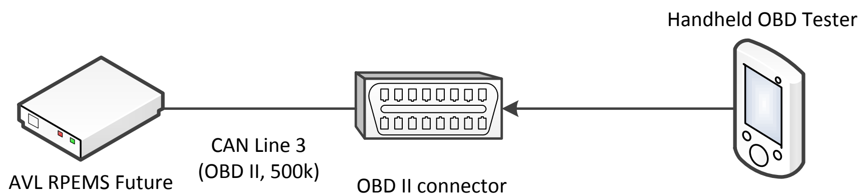


Figure 3-4: OBD II diagnostics system overview

Hardware Cost (per unit): low (generic OBD II tester)
Software Cost (per unit): none (included in hardware cost)
Development effort: high (OBD software module)
Maintenance effort: medium-low (maintain OBD software module integration)
Ease of use: high (every available scanner works, no specialized knowledge needed)

3.5 Rating matrix

The concepts mentioned in 3.1 - 3.4 are summed up in Table 3-1, with the rating criteria listed for each entry converted tokens, from ⊖⊖ (worst) to ⊕⊕ (best).
With a token sum of +2, the OBD implementation gets the highest overall rating, with the low per-unit cost as one decisive factor, user-friendliness being the other one.
Therefore it was decided that the diagnostic solution for the AVL RPEMS should be an OBD II implementation.

Table 3-1: Concept rating matrix

	present scenario	external logger	special software	OBD II implementation
SW modifications required	-	add module for CAN export of variables	add modules for persistent storage and CAN data export	add modules for persistent storage and OBD II communication
Hardware required (per RPEMS unit)	ETAS compatible CAN PC-CAN Interface (e.g. ETAS ES580 PCMCIA Card)	Data logger wiring	CAN – PC Interface	OBD compatible tester / PC-CAN interface
Software required (per RPEMS unit)	ETAS INCA compiled ECU Firmware files (HEX + A2L)	data analysis software / driver software	Custom Analysis Software	none (hardware tester) or diagnostic software
Documentation required for diagnostics	complete ECU firmware documentation	complete ECU firmware documentation	Depending on Software features	none
HW Cost	⊖⊖	⊖⊖	⊙	⊕
SW Cost	⊖⊖	⊙/⊕	⊙	⊕⊕
Development	⊕⊕	⊕	⊖⊖	⊖⊖
Maintenance	⊕⊕	⊕	⊖⊖	⊖
Ease of Use	⊖⊖	⊖	⊕⊕	⊕⊕

Chapter 4

Design

This chapter covers the environment the OBD II software should be integrated into, meaning aspects of the RPEMS hardware, toolchain and existing software that influence the implementation described in Chapter 5.

4.1 Specification AVL RPEMS Future

The AVL RPEMS Future is the successor to the RPEMS NG (Next Generation) rapid prototyping ECU and has been developed to fulfil the needs of modern GDI engine concepts.
Its core is a System on Module (SoM) produced by Phytec based upon Infineon's TriCore TC1797 (phyCORE-TC1797), which is mounted on a proprietary motherboard. The motherboard contains the power supply, the required power drivers, signal conditioning, interface components and a number of peripheral devices. The only periphery relevant to this thesis is the ST M95640-W type EEPROM, which is connected to the SoM by SPI, providing 64kBit of nonvolatile memory.
The main specifications of the phyCORE SoM according to [14] are:

- 180MHz MCU clock
- Up to 8Mbyte SRAM
- 64MByte Flash
- 2 x I^2C controllers
- 2 x SPI controllers
- 2 x RS232 UART with one UART/USB converter
- 4 x high-speed CAN interface
- on-board 3,3 V regulators and reset controller

Of the 10 communication channels listed above, the four CAN interfaces and the USB interface of the USB / RS232 converter on the first UART are routed to the ECU main connector.

4.2 Requirements for the OBD II implementation

As presented in Chapter 2, OBD II's primary focus is monitoring emission-related systems, which is not the primary objective here. Instead, the focus lies on more general diagnostic functions, especially:

- Displaying sensor data and other internal values
- Non-volatile incident memory
- Recording boundary conditions to incidents
- Identifying the software/calibration version.

For these reasons the following OBD services were chosen to be implemented:

- Service 01_H – live data
- Service 02_H – freeze frames
- Service 03_H – DTC storage
- Service 04_H – clear DTCs
- Service 09_H – query vehicle information

Since the software package for each RPEMS project is unique (software modules in use, project specific versions of software modules) the OBD software should be as self-contained as possible with minimal changes to the environment required. Ideally, it should only be necessary to import the OBD package into an ASCET library and assign its processes to the scheduler to make it work.
The Implementation does not have to be fully OBD II compliant, except for the communication with the external test tool as it should support any OBD II compliant tester.
Since cooperative multitasking is employed, it is important that the software components are non-blocking and take up as few CPU time as possible.

4.3 Toolchain

The toolchain used to build the RPEMS software is illustrated in the diagram below:

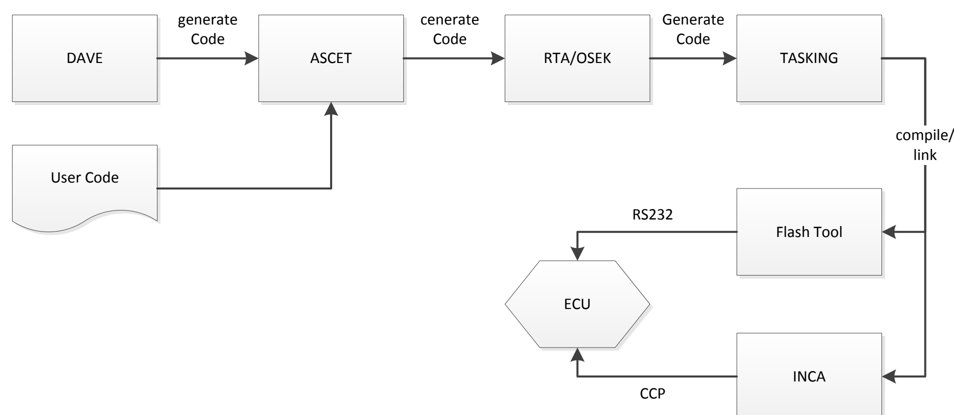


Figure 4-1: AVL RPEMS toolchain

DAVE is short for Digital Application Virtual Engineer and is a code generator for Infineon Microcontrollers provided by Infineon. It is designed to help with the generation of driver and initialization code and is the source of many of the low-level functions inside the RPEMS Firmware.
ASCET organizes software projects in „databases“. Databases contain a tree structure with all the software modules inside the database as well as “projects”. Projects contain several of the software modules inside the database and configuration information that is used to build the project (Figure 4-2).

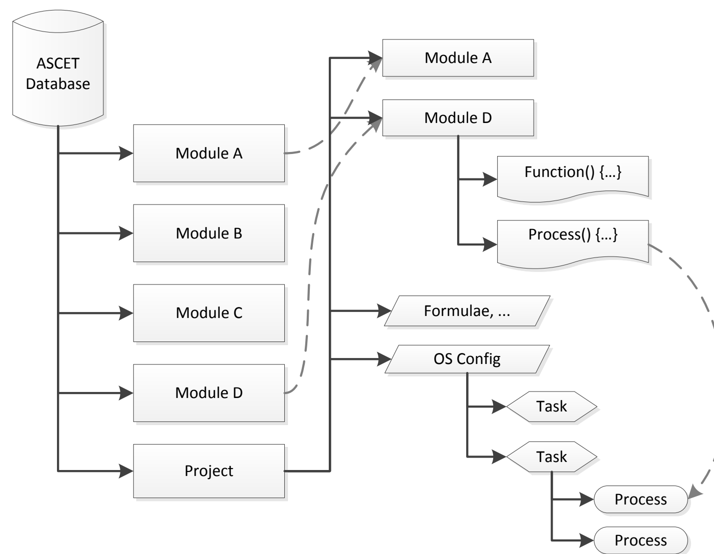


Figure 4-2: ASCET component hierarchy

ASCET supports several different ways of implementing software. The most basic is to write "raw" C, where no checks additional to those done by the compiler are performed. However, even in "C" mode, the user code is just the function body and part of a larger, generated, file that is compiled. ASCET adds the function headers and variable definitions during code generation to make the implementation compatible with the rest of the environment.

Other options of code generation are block diagrams, state diagrams and ESDL, which is a programming language with a Java-like syntax.

OSEK is a European automotive industry standards effort to produce open system interfaces for vehicle electronics. OSEK is an acronym formed from a phrase in German, which translates as "Open Systems and Corresponding Interfaces for Automotive Electronics".

The goals of OSEK are to support portability and reusability of software components.

RTA/OSEK is an implementation of a Real Time Operating System based on OSEK standards, namely AUTOSAR OS SC1 and OSEK/VDX OS V2.2.3. The Code that is generated by it is also MISRA-C compliant. Its kernel manages task execution, -switching and -communication and provides optional instrumentation for debugging and timing analysis.

For this project, Version 5.0.1 has been used.

Tasking VX for TriCore (Version 3.3r1) is used as the macro assembler, C compiler and linker. It also provides static and runtime libraries to the compiler.

ETAS INCA is a calibration tool that allows reading and writing of internal data (variables, maps, characteristic curves) on a connected ECU. The communication with the ECU is carried out via Can using the CCP (CAN Calibration Protocol) standard. Since the protocol works by directly accessing and manipulating the ECU's memory, INCA needs the compiled

firmware (.hex file) and a ASAM2 descriptor file containing type information (.a2l file), which are generated during the compile process.
The "Flash Tool" Software is a small PC Software tool to load compiled firmware onto the Flash Memory of the TriCore MCU used in the RPEMS. This is done via a serial connection to the target and is even possible if the firmware currently running on the MCU is non-responsive.
If a suitable firmware supporting CCP is already flashed on the MCU, it is possible to re-flash via CAN connection using CCP and INCA on the PC side.

4.4 Current Architecture of the RPEMS FUTURE System

4.4.1 Task scheduler

When starting with a fresh ASCET project for the RPEMS target the only functionality present is the one provided by the RTA kernel, the most important of which is the task scheduler.
The configuration of the scheduler is directly possible via a GUI provided in ASCETSs project editor. The editor is used to define tasks for the scheduler and to assign processes to tasks.
Typically, there are four types of tasks present in a project:

- Init tasks
- Alarm tasks
- Interrupt tasks
- Software tasks

The **Init task** is called upon startup and typically contains hardware initialization code. It offers no scheduling options since it is only called once before the actual scheduling starts.
Several **Alarm-type tasks** fire periodically at a selectable frequency. Typically there are tasks present with a period of 0.25, 0.50, 1, 10, 50 and 100 milliseconds. The scheduling mode is set to cooperative and the highest-frequency task carries the highest priority. All of the application code processes are assigned to these tasks.
Hardware interrupts are handled by Interrupt-type tasks. Each task is assigned an interrupt vector and has a selectable priority. Usually only one process, the interrupt handler, is assigned to an interrupt task although it is possible to have multiple processes called by the interrupt.
The "sync" task is a **software task**, which means it is can be activated by a system call somewhere in the application software. The "sync" name is derived from the fact that it is called synchronous to the revolutions of the combustion engine that is controlled by the RPEMS. This mode of operation is important for cycle-synchronous processes like injection, ignition and knock and misfire detection.

4.4.2 Drivers

Drivers and the Hardware Software Interface / HAL in general are realized as regular software modules. The low-level hardware interface that deals directly with the control registers of the microcontroller is typically comprised of two parts. One part is code generated by Infineon's DAvE, which provides appropriately named wrapper functions for accessing the hardware registers and initialization functions. The second part uses various processes that reference and call these functions, usually an init process and several cyclic worker processes.
On top of the driver software there is a remapper process that connects the variables provided by the low-level driver to application-specific variables performing data conversion if necessary.

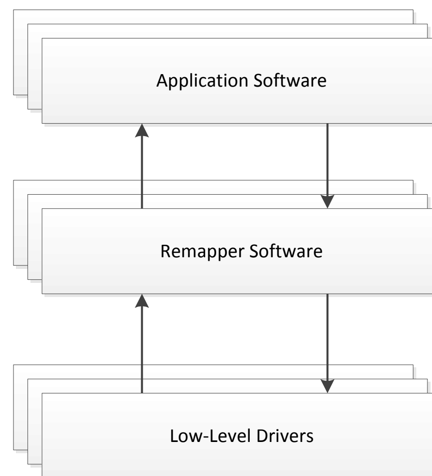


Figure 4-3: RPEMS driver structure

As an example, this is (a simplified version of) how an ADC driver would work:
The low-level driver process is called by the ADC interrupt task and copies the ADC conversion result register value into global ADC value variables. A periodically called second process sets the ADC up to capture another sample, therefore setting the sampling frequency.
In a cyclic process the remapper uses the ADC result variables and characteristic curves to calculate a physical value represented by the digitized value, e.g. temperature or throttle position and stores the result in another global variable to make it available for e.g. the load calculation process.
The advantage of this architecture is a high degree of modularization and interchangeability. If the source of a certain signal changes, (different ADC channel or from ADC to CAN) only the remapper(s) have to be adopted, the application software and drivers remain unchanged. This makes the application software relatively hardware-independent increasing its portability. However, this model of complete abstraction is not ideally suited for all

applications. In chapter 5.1.1 it is shown how the CAN driver has to be modified alongside with the remapper to provide the functionality needed for the OBD application software.

4.4.3 Data Consistency

With a multitude of processes accessing shared data elements, also known as concurrency [15], the aspect of data consistency has to be considered, especially in interrupt-driven systems, where the execution order cannot be predicted during code creation. A common problem occurring on such systems is that a shared variable is read in multiple locations in the same process, expecting the result to be the same while an interrupting process changed the variable, leading to errant, nondeterministic behavior. This type of situation is known in programming as a "race condition", defined by the following three properties: [16]

- Concurrency – at least two control flows are executing in parallel
- Existence of a shared property – a shared "race object" is accessed by the concurrent control flows
- Changing of state – at least one of the concurrent control flows alters the state of the race object

Hence the name race condition, because two processes "race" for access to the race object, the timing of which determines the outcome of the control flow as illustrated in Figure 4-4.

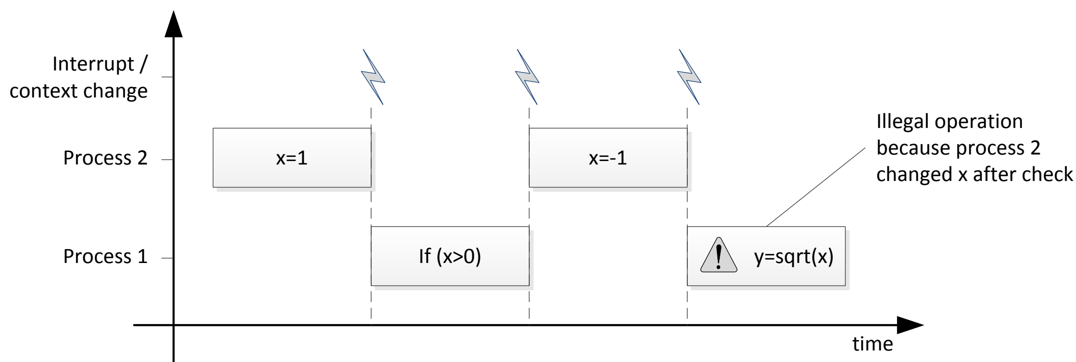


Figure 4-4: Data becoming inconsistent as result of a race condition

A possible remedy for this situation is, to create working copies for all shared variables at the beginning of a process and use these private copies for all operations, as demonstrated in Figure 4-7.

This solution to data consistency is implemented by protected global variables, so-called "messages", in ASCET. When using variables declared as messages, copies are automatically created for each receiving process.

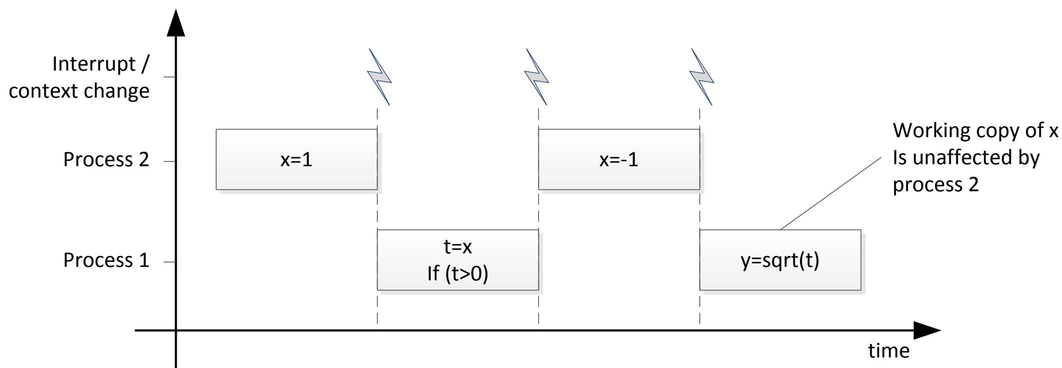


Figure 4-5: Avoiding the race condition

The message system, however, has a few limitations

- Messages cannot be used in classes
- Messages cannot be used for non-scalar types (arrays, …)
- Messages only offer protection in 1-to-n relations (1 sender, n receivers), for the case of multiple senders additional resource management is needed to avoid race conditions leading to the “lost update” problem [17], when concurrent processes try to change a shared variable.

Messages are used in many of the existing modules of the RPEMS software, although not exhaustively and consistently. Why this does not lead to problems is apparent when looking at the scheduling algorithm of the RTA/OSEK kernel, taking into account that all processes in the RPEMS software are defined as cooperative.

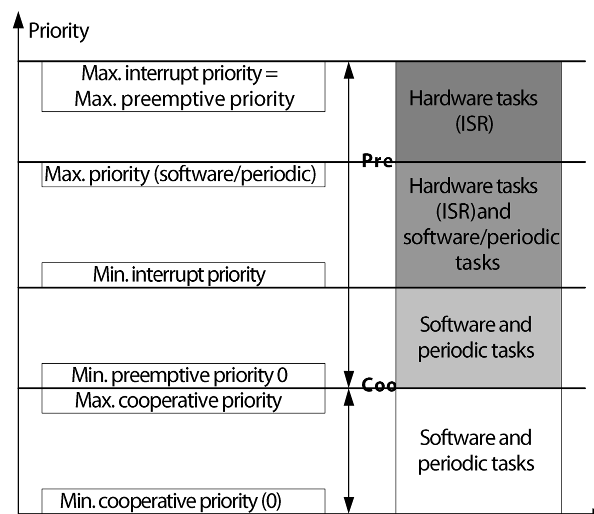


Figure 4-6: RTA task priority scheme[19]

While cooperative tasks do have the lowest priority, the processes spawned by them will not be interrupted by of a task with higher priority. The scheduler will wait until the cooperative process has finished and interrupt the task to run the higher-priority tasks pro-

[19] Taken from [30]

cess(es), then returning to the lower-priority task. In the example below, processes p1 and p2 belong to Task 1, while processes p3, p4 and p5 belong to Task 2:

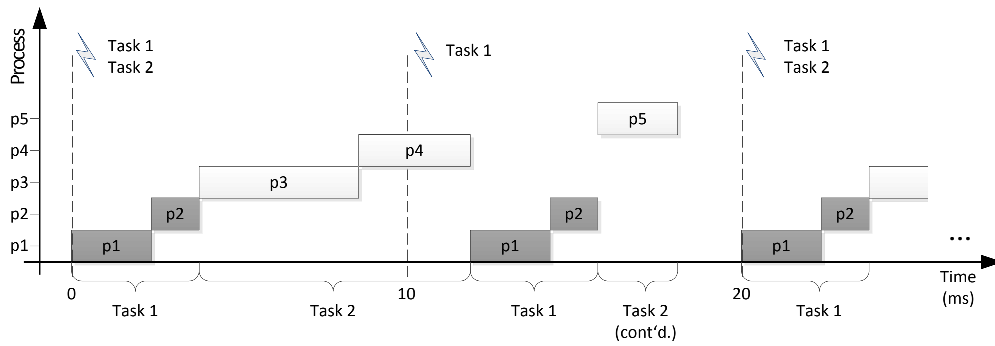


Figure 4-7:cooperative scheduling example

While the order in which processes are executed is not predictable on this system, each process relies on two paradigms:

- A process will not be interrupted by another process, except for interrupts.
- For each task, the processes will be executed in-order, e.g. p1-p2-p3

Since the interrupt-driven driver software is abstracted by scheduled remappers, interrupt processes are also of no concern to the application layer software, only remappers need to be aware of the possibility of being interrupted.

Chapter 5

Implementation

This chapter describes the structure and functioning of the OBD II software modules as well as the changes that had to be made to existing RPEMS software

5.1 Modifications to existing Software

While one of the design aims of the OBD II software was to keep it as self-contained as possible, some changes to external software were unavoidable. The CAN driver stack had to be extended with a handshaking mechanism while the EEPROM driver had be configured to allocate additional memory.

5.1.1 CAN Software Stack

Like the other I/O drivers, the CAN Software interface is also composed of a "Low Level" driver and a remapper.

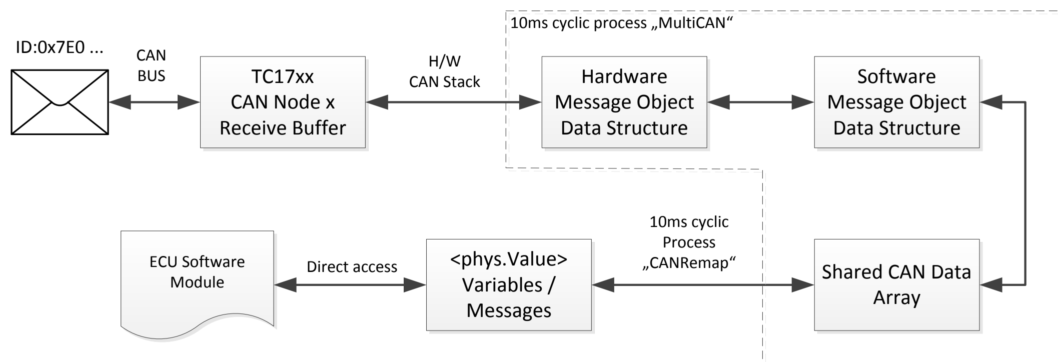


Figure 5-1: Block diagram of the CAN stack

The TC17xx integrates four hardware CAN interfaces ("nodes") and 128 memory-mapped CAN Message object structures ("MObs"), that can be dynamically assigned to the interfaces. By setting the ID, acceptance filters and status registers of the MObs, the nodes are configured to listen to the messages their MObs match.

Upon arrival of a new CAN Message, the node that accepted it copies it from the receive buffer to the "data" field in the corresponding hardware MOb structure.

The "MultiCAN" process periodically copies the contents of the hardware registers of all MObs configured to receive data into Software MObs. From there MultiCAN copies the contents of the data field to shared (global) data arrays.

The “CANremap” process periodically takes the array data and copies it into the appropriate internal ECU variables and messages, thereby being the node where the association between CAN Data and internal data takes place.
For Internal data that is to be put on the CAN bus, the process works in the exact reverse order with the exception that MultiCAN also sets a status register of the corresponding hardware MOb to indicate the message as “to be sent”.
This correlates to the producer-consumer software pattern used in object-oriented programming, where the application software and MultiCAN conduct the roles of producer and consumer and CANremap acts as the pipe.
Using this pattern without a flow control mechanism may result in problems depending on program timing. [18] The way existing remappers take care of this problem is to act as what can be described as a single-element LIFO (Last-In-First-Out) buffer, discarding all but the last element sent by the producer, before the consumer is ready to process the next element. This solution is viable, since there is no need for synchronization between producer and consumer and producer data lost between two runs of the consumer, does not matter, as long as the consumer gets recent data.
The protocol side of OBD II, on the other hand, is entirely dependent on data only being sent and received in a certain order and a specific number of times (once). Therefore the CAN stack has to be modified so that end-to-end information about the status of CAN data (received / read / sending /sent) can be conveyed.
This is accomplished by implementing a simple flow-control mechanism, using binary semaphores as suggested in [18].
In the receiving direction, the “NEWDAT” bit provided by the hardware CAN stack for each MOb has to be monitored. It is set whenever the CAN controller copies a new message from the receive buffer into the MOb and has to be cleared by software. So whenever MultiCAN sees the NEWDAT bit set, it should copy the data into the receiving array and then clear the bit to mark it as “read”, while setting a “new data” flag visible to CANremap/OBD II.
In the transmitting direction a software solution has to be implemented in a way that the OBD II software can indicate that a message has to be sent. MultiCAN copies the content to a HW MOb, starts the transmission by setting TXRQ and clears the “ready to send” flag which has in the meantime blocked the OBD II software from overwriting its own data before it could be sent. MultiCAN then monitors TXRQ and does not overwrite the HW MOb before it is cleared, indicating a successful transmission.
This establishes a simple flow control mechanism as depicted in Figure 5-2. Note that while CANremap is shown as transparent, the information is really passed on through different variables.

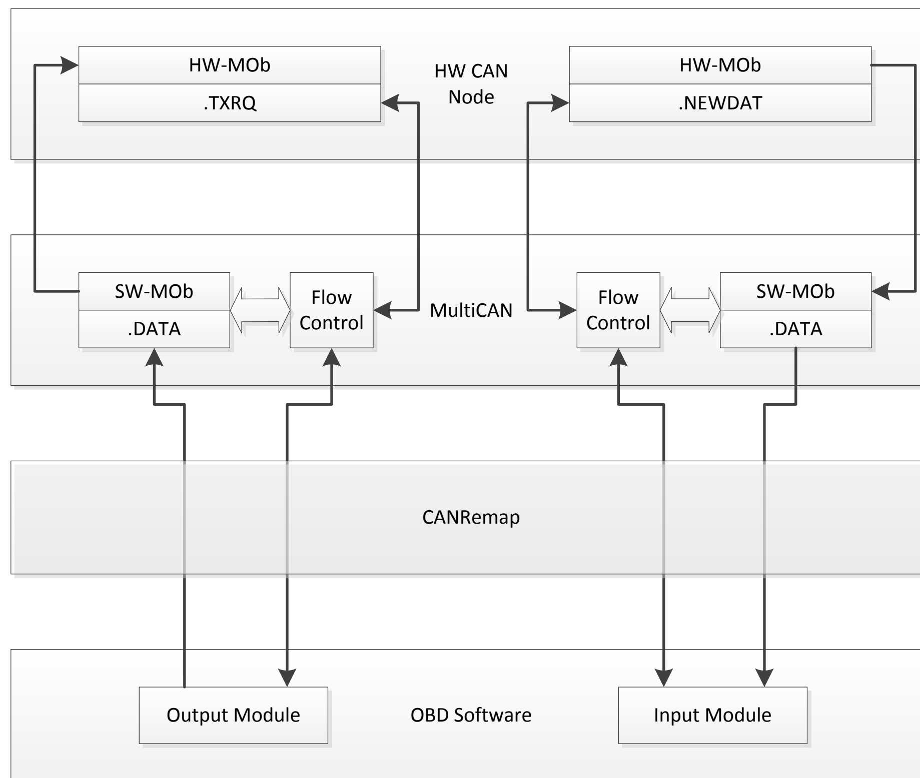


Figure 5-2: Block diagram of the modified CAN stack

As mentioned in chapter 4.4.2 this modification breaks with the idea of complete abstraction through remappers as the application software requires specific changes to be made to both underlying levels. This is however, done deliberately for two reasons:

- The changes to the low-level CAN driver take up very few resources and could be seen as an effort to make it “OBD-ready”. If the OBD functions are not used, the overhead would be just two unused MObs, a few bytes of variables and a couple of “if” statements which never branch.
- The CAN driver could be modified so that all MObs support handshaking. However, since there are 128 Mobs, a few bytes each would accumulate to a considerable overhead over the original driver while the mechanism is unlikely to be used by application software other than the OBD server.

5.1.2 EEPROM Driver

The EEPROM driver was built as an extension of the SPI driver developed in [19] and has so far only been used in an proof-of-concept application. The driver provides access to the EEPROMs by means of pages, each 16 bytes wide containing 14 bytes user data and two command/control bytes.

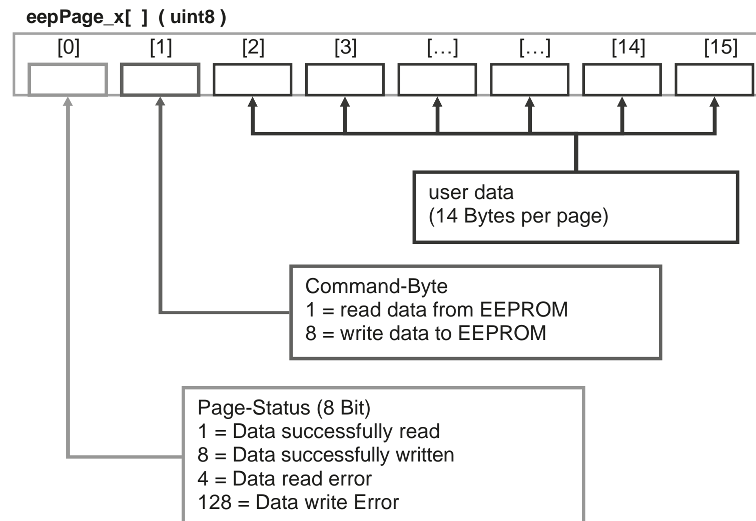


Figure 5-3: EEPROM page structure[20]

Pages can be either static or dynamic:

- Static pages are read to RAM from the EEPROM at startup and written back on request. Accessing them causes neither overhead nor delays, since all read and write operations take place in RAM. RAM expense is identical to the page total.
- Dynamic pages work by mapping one page in RAM to different physical EEPROM pages by setting eepDynAdr (the EEP dynamic address). Both writes and reads (especially across page borders) experience delays since both transfers have to be requested first and not till after completion of the EEP/SPI cycle the next physical page can be addressed. RAM expense is constant at the size of one page.

Since the OBD II persistent storage module will need to access a memory area much bigger than a single page, reading on demand is out of the question for performance reasons, since the read would at least take $(n_{pages} + 1) \cdot 100ms$. It would be possible to read the pages at startup and write them back in the background but that would not offer a memory advantage over static pages and would also mirror part of the functionality in the EEPROM driver.

Therefore the use of static pages has been chosen as memory access strategy.

The modifications to the EEPROM driver were thence limited to adding more pages to the multiplexer in the page selection logic.

[20] translated from the EEPROM driver documentation

5.2 New Software Modules

5.2.1 Important Shared OBD variables, constants and calibration tables

A number of data elements are shared among multiple modules of the OBD software. The most important ones are briefly introduced here to aid the understanding of their usage in the individual software modules:

- EOBDPIDData[] ("Live Data Array") is an array of 32-bit wide elements indexed by PID number that stores the current value of the supported PIDs
- EOBDPIDSize[] ("Size Info Array") is indexed by PID number and contains information about how big (how many bytes) the representation of each PID is. This information is taken from SAE J1979DA:2011 table B2 ff.
- EOBDPIDsAvailable[] ("Service 1 Support Array") is indexed by PID number and contains a bit value that has to be set for each PID which should be supported by the ECU. This is used to build the 00_H, 20_H, ... PIDs
- EOBDFFPIDs[] ("Service 2 Support Array") stores the PIDs that should be saved when a freeze frame is captured. While the PIDs have to be arranged consecutively in the array, they do not need to be in a certain order. This array is a parameter and can therefore be calibrated at runtime.

5.2.2 The OBD State Machine

The OBD state machine (Figure 5-4) implements the core logic of the OBD protocol. While it offers no functional advantage to implement as a block diagram in comparison to C, it is more comprehensible and offers an easy entry point for future modification and expansion.

Upon startup ("idle" state), the state machine listens for new CAN Messages with its functional or physical ID (see Table 2-6) and looks into the message to identify the SID. Then it changes its state to invoke the appropriate SID handler on the next execution cycle. Some of the handlers complete in a single cycle and immediately reset the state to "idle", others require multiple execution cycles (multi-message replies) and only reset upon completion. To avoid lock-ups due to e.g. protocol errors, these handlers include internal time-out mechanisms that reset the state machine to "idle" if they get stuck.

All diagrams associated with the OBD state machine can be found in Appendix 4: Block Diagram 1ff.

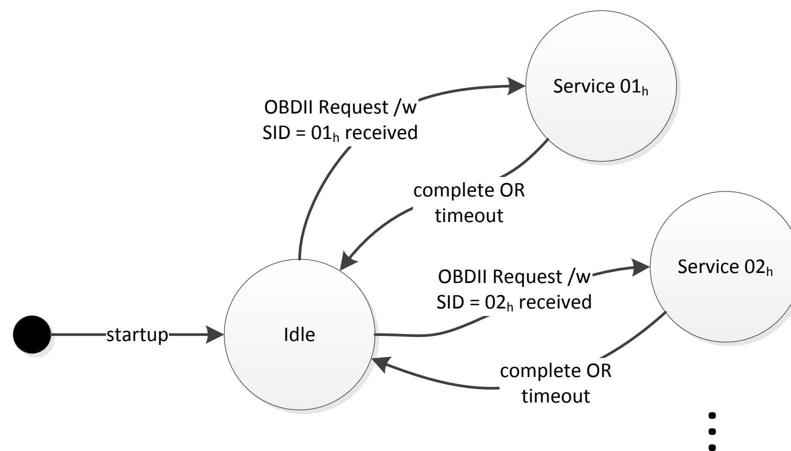


Figure 5-4: main OBD state flow diagram

5.2.3 The OBD Remapper

As mentioned in chapter 4.4.2, the software architecture of the RPEMS uses the concept of remappers to abstract hardware registers and driver variables from variables used in the application software. This offers not only the advantage of a more structured design; it also enables the use of ASCETs automated conversion between the physical and the implementation domain.

The physical domain in this context means the physical value/meaning of a variable (e.g. 21°C, 31%) while the implementation describes how the value is stored in memory. The relation between the two defines the limits to the physical values, which can be stored as well as the quantization and thus the resolution of the data.

These relations are stored in the ASCET project as “formulas”, using the $f(x) = \frac{a+b \cdot x}{c+d \cdot x} + e$ notation as the most complex form, other options being linear ($f(x) = a \cdot x + b$) and “identity” (1:1). [20]

One aspect that has to be considered when using formulas is that when used in block diagrams only the physical value of a variable can be accessed, while using a variable in C code only returns the implementation value.

These characteristics of the ASCET variable model are used in the EOBD remapper. In it, the internal ECU variables, which are to be used in the OBD II module, are copied to the OBD variables using block diagram specification. Because the OBD variables have been associated with formulae that implement the value quantization defined in SAE J1979:2011 table B2 ff., the implementation value of the OBD variables is identical to the binary value that needs to be loaded in the CAN packets during ISO 15031 communication.

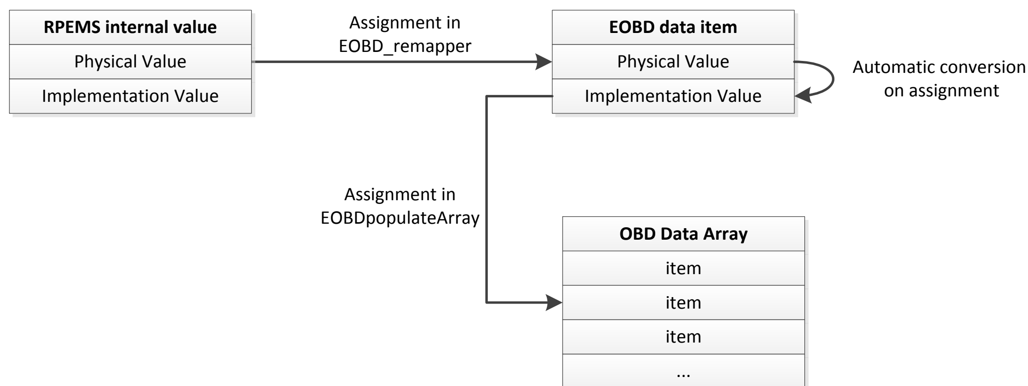


Figure 5-5: usage of the ASCET variable model by the OBD remapper

From the more common PIDs (< 40_H) some are not representations of ECU internal values, but contain special information about the OBD service. One that is supported by this implementation would be PID 01_H, which contains information about the number of stored DTCs and the state of the MIL and various OBD tests and is calculated directly in the remapper (see Block Diagram 11: EOBD_remapper EOBD_01_DTC_CNT subdiagram in the appendix)

Another type of special PID are 00_H, 20_H, 40_H ... which bit-encode the support for ranges of 32 PIDs each. Since the support for certain PIDs does not usually change during runtime, the bit patterns for these PIDs are calculated from the Service 1 Support Array during initialization by the EOBDpopulateArray::update_supported_PIDs_Mode1 function.

Since each of the bitfields encodes if the next one is supported, this is done by walking the array from end and only marking non-zero bitfields as supported. The full implementation can be found as Listing 3 in the appendix.

To enable the other OBD Software modules to access the PID Values by their number (and therefore to enable any kind of generic response) the PID values have to be stored in the Live Data Array. Since the value that needs to be stored is the implementation value this needs to be done in C. The following code snippet represents what executed by EOBDpopulateArray::updateEOBDlivedata_100ms – see Listing 5 in the appendix for the full code.

```
001 #define EOBD_VAL(PID, VAL) EOBDPIDData[PID] = VAL << ((4-EOBDPIDSize[PID]) * 8);
002
003 // EOBD_VAL(0x00, 0);
004 EOBD_VAL(0x01, EOBD_01_DTC_CNT);
005 // EOBD_VAL(0x02, 0);
006 // EOBD_VAL(0x03, 0);
007 EOBD_VAL(0x04, EOBD_04_LOAD_PCT);
008 EOBD_VAL(0x05, EOBD_05_ECT);
009 // EOBD_VAL(0x06, 0);
```

Listing 5-1: Loading PID Values in the PID Array

5.2.4 The OBD Persistent Storage Interface

Code fragments (C, ESDL, block diagram) can be in one of two containers: a function or a process.

- Functions can take arguments and provide a return value. They cannot be attached to tasks.
- Processes do neither take arguments nor provide return values, but they can be attached to tasks.

Each executable module (C, ESDL, block diagram) may either be a regular or a class module.

- Regular Modules can only be instantiated once in a project, which means that function calls can be made by at most one other module. Usually they instantiated at the project root and called only by the scheduler.
- Class modules can be instantiated multiple times, e.g. by different modules. Although the generated code is C, the framework emulates the behavior of C++ class members with local variables becoming instance variables and public (ASCET: "exported") variables become class variables. ASCET classes do, however, not exhibit the other traits of C++ OOP classes like inheritance or derivation.

The DTC Storage Interface Module (EOBD_DTC_Interface) provides access to the EEPROM-backed nonvolatile memory areas where DTCs and freeze frame data are stored.
The module should be integrated into each software module that needs to store DTCs as well as in the OBD software to retrieve them. This mandated the use of a class module (multiple instances) and the use of functions (arguments and return values).
The Storage Interface Module provides the following (public) functions

- Init() performs some one-time tasks to initialize the memory. It only has to be called once by only one instance.
- EOBD_DTC_MGMT_100ms checks if the EEPROM memory is yet available. When it is, it restores state variables from it and generally enables memory access.
- StoreDTC() adds an entry (DTC) to the fault memory, which can be read by service 03_H. The fault code can be specified as well as if the MIL should be turned on and if a Freeze Frame should be stored. While StoreDTC will not store duplicate DTCs, it will create multiple freeze frames for the same DTC if requested.
- ClearDTCs() deletes all stored DTCs and Freeze Frames. It is called by service 04_H.
- getFFPID() returns the data of a certain PID in a Freeze Frame.
- DTCMessageStream() provides an ISO 15765-2 compatible (see 2.4.6) stream of CAN messages for service 03_H.
- commitToEEP() makes the module write its nonvolatile memory back to the EEP.

All C sources associated with the persistent storage interface are included in Appendix 5: Listing 7: EOBD_DTC_Interface implementation code

Memory Organization

During the initialization phase, EOBD_DTC_Interface loads an array of the type uint8*[] (equal to a uint8**) with the array pointers (uint8*) to the EEPROM pages reserved for this use. This way the available memory appears as a flat structure that can be programmatically accessed as a 2D array even if the EEPROM pages are non-consecutive.
The example implementation uses 9 EEPROM pages with a total of 126 bytes of memory available as storage. Using constants defined in the C header, the memory is segmented like this:

Page Number	Page Byte: 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	Status byte	Command Byte	Number of DTCs	Number of FFs	first DTC	MIL status	first FF									
2			DTCs													
3																
4																
5																
6			FFs													
7																
8																
9																

Figure 5-6: OBD persistent storage memory organization

The DTC storage shows the following characteristics:

- DTCs take up a constant space (2 bytes each).
- DTCs are only cleared as a whole, so memory fragmentation is not possible.
- When the DTC memory is full, it wraps around and the oldest DTC is overwritten.

This mandates, that two values are kept in the nonvolatile memory: the number of DTCs stored and the position of the oldest one (first DTC). Also, since the memory size is always a multiple of the DTC size, the full memory can be utilized, creating a ring-type memory.
The FF storage has the following characteristics:

- The space a FF takes up is dependent on the PIDs that are included
- FFs are only cleared as a whole, so memory fragmentation is not possible
- When the FF memory is full the oldest FF will be overwritten.

As with the DTCs the number of stored FFs and the index of the oldest one have to be stored in nonvolatile memory. Since the composition of a FF can be calibrated via the Service 2 Support Array, the size and maximum number of freeze frames is variable and recalculated on startup and only stored in RAM. Since the memory size is likely not a multiple of the FF size, there is always some unused memory (worst case: $sizeof(FF\ memory)/2 - 1$ is unused when one FF takes up just over half the available memory). The rest of the memory creates a ring with $floor(sizeof(FF\ memory)/sizeof(FF))$ elements.

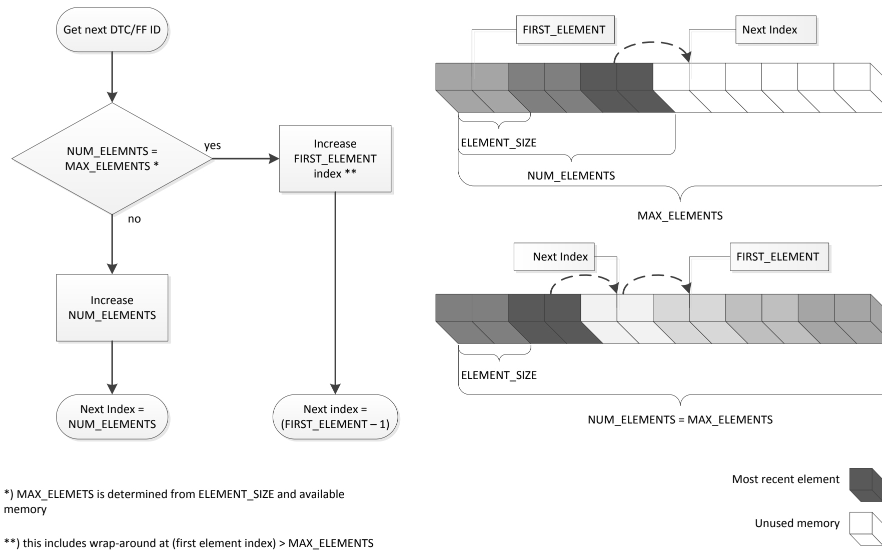


Figure 5-7: persistent storage element allocation logic

Clearing the DTC & FF memory

To clear the persistent memory, resetting the pointers to the start and zeroing the element counts is sufficient. If new elements are stored, the old ones will be overwritten. To overwrite the complete FF and DTC memory, a calibration parameter (EOBD_DTCFullerease) can be set.

In contrast to the other functions dependent on the EEPROM Memory, this one does not check the memory availability. This is done externally in the service 04_H logic to enable the service to abort and send a negative response (see Memory Organization) if it is not available.

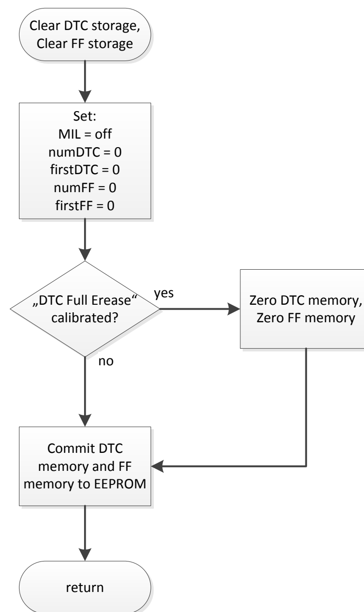


Figure 5-8: "Clear Memory" Program Flow

Storing and retrieving a DTC

When a new DTC should be stored, the code checks the DTC memory if that DTC is already there to keep a reoccurring error from flooding the DTC memory. If the DTC is not found, the index of the next available DTC slot is requested from the memory management (see Memory Organization) and then translated in an address (page number and offset) in the DTC memory block. Subsequently, the 16-bit DTC is split into two bytes and written at this location.

The MIL request is honored even if the DTC is a duplicate.

Finally, commitToEEP() is called to save the changed RAM content to the EEPROM.

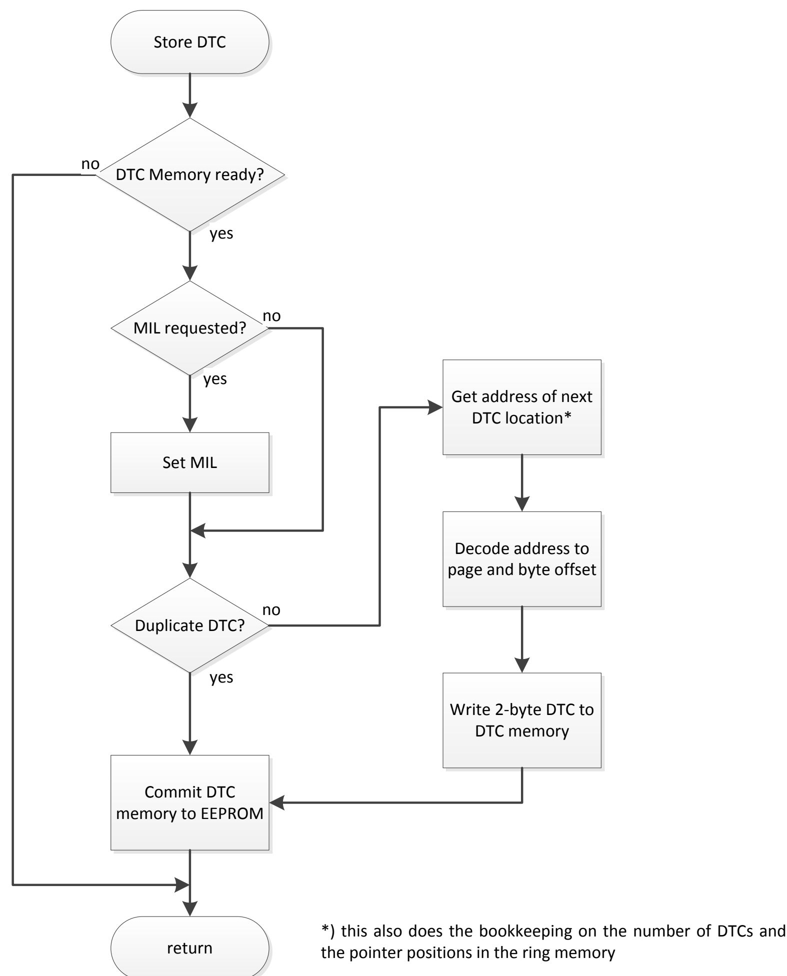


Figure 5-9: "Store DTC" Program Flow

To retrieve a certain DTC from the fault memory, first the absolute position of the DTC is determined in the ring memory (may be different to its relative position due to overflowing of the ring see Figure 5-6) and then the page number and offset are requested from memory management.

Finally, two bytes are read from this location and assembled to form the 16-bit DTC.

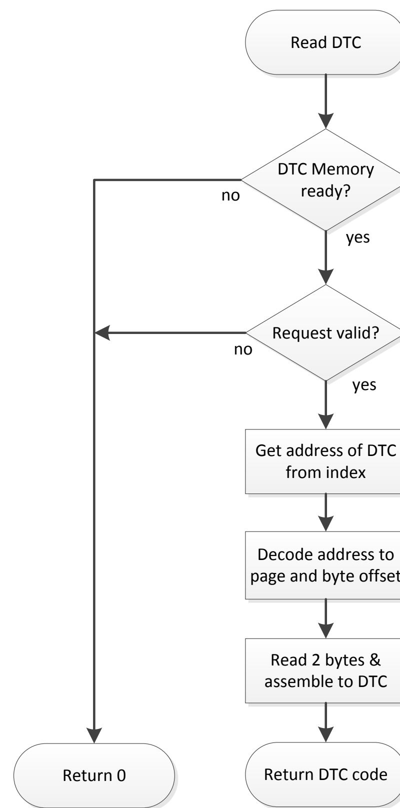


Figure 5-10: "Read DTC" Program Flow

Deferred DTC storing

While slow calls to e.g. CAN and EEPROM drivers have been avoided, writing to the fault memory can consume significant time, especially compared to the system's fastest timebases (0.25 to 1ms). To avoid deadline misses if the DTC interface is to be used in a module that runs on such a quick timebase, storing of DTCs and FFs can be deferred (see Figure 5-11).

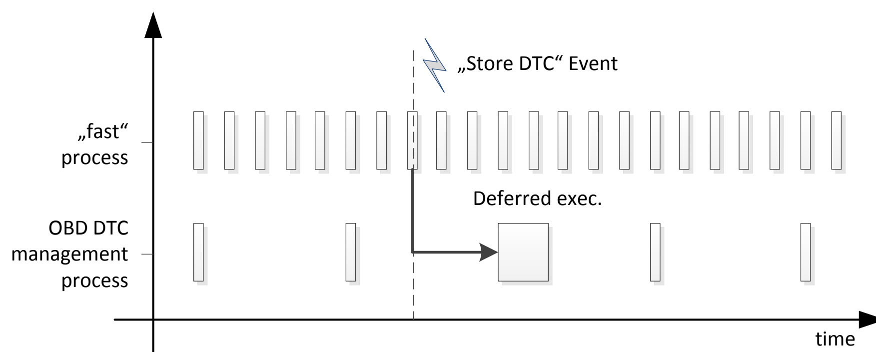


Figure 5-11: deferred DTC storing

To do so, the function StoreDTCdeferred() can be called, which takes the same arguments as the regular Store DTC function but just copies the arguments to global class variables, what causes the management function running in the 100ms grid to take over actually

storing the DTC or FF. This is not without disadvantage: while the up to 100ms offset in the FF data are negligible (the PID data is only refreshed in 100ms intervals, anyways), the function is prone to race conditions causing "lost data" effects. This happens when StoreDTCdeferred() is called multiple times between two executions of the management function, where the DTC in the arguments of the last call will be the only one actually stored with the rest discarded.

Storing and retrieving Freeze Frame Data

A freeze frame is a snapshot of a number of PIDs and the DTC that caused the freeze frame to be captured. The PIDs included in a freeze frame are a subset of the ones available in service 01_H and are set via the service 2 support array. To determine the total size of a freeze frame and thus the number of FFs that fit in the available EEPROM memory, this array is walked and the corresponding entries from the PID size Array are added together.
When a new FF needs to be stored, the function first requests the ID of the next available FF from memory management and then requests the memory address of the start of the FF with this ID. Then first the DTC is written, followed by all of the PIDs listed in the service 2 support array. After each byte, the memory address is incremented and retranslated in a page number and offset, since the PID boundaries do not necessarily match the page boundaries.
To retrieve a PID from a stored freeze frame, the PIDs location in the memory blob, containing the FF, has to be determined. Since the PIDs are stored in the same order they appear in the service 2 support array, this array is walked, adding up the PID sizes from the PID size array, until the PID is found in the service 2 support array. Then, using this byte offset and the FF ID, the memory location of the PID can be requested from memory management. Finally, the PIDs data is read and assembled to give the PID value.

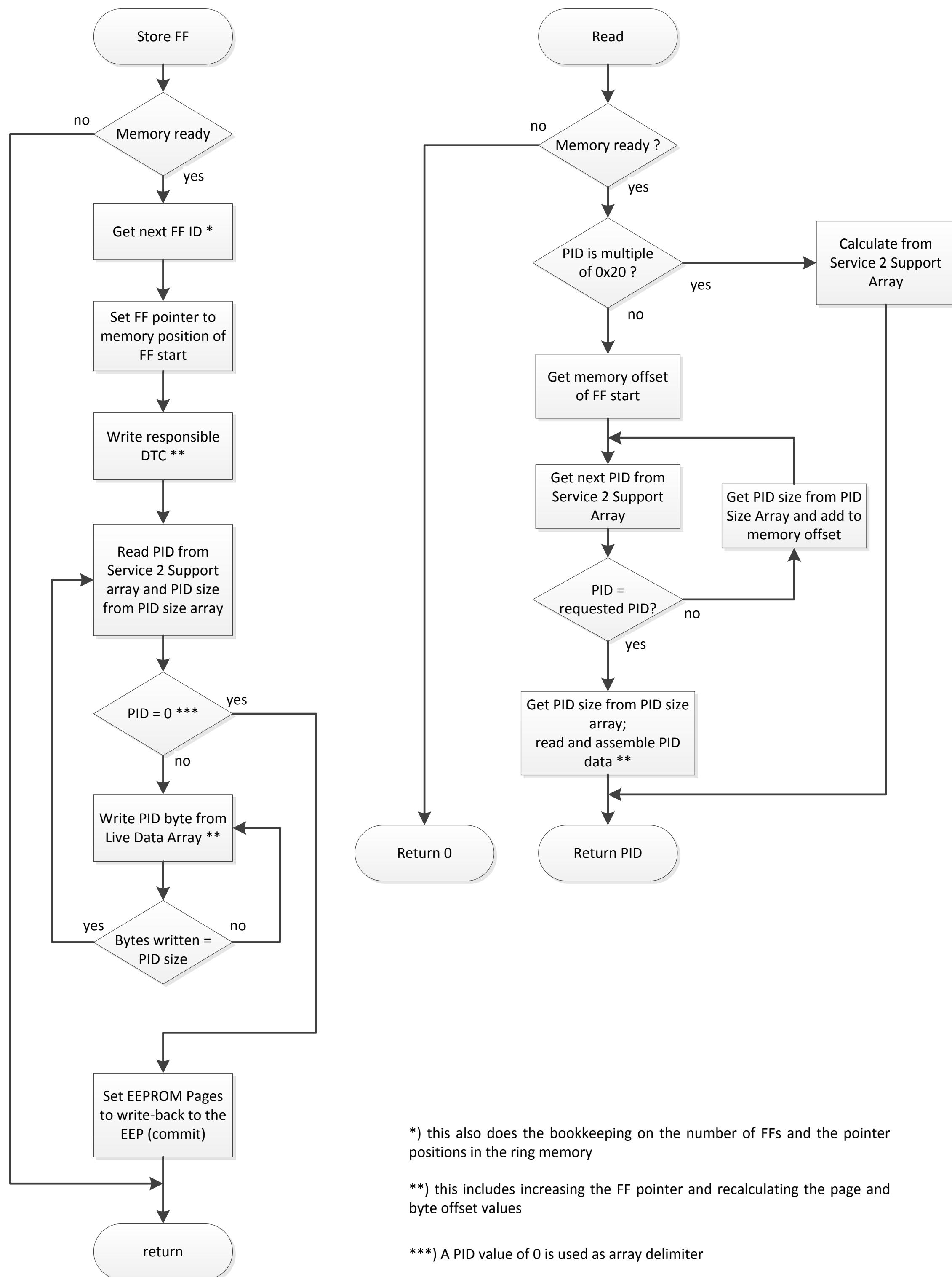


Figure 5-12 : Program flow for storing and retrieving FF data

Answering a Service 03_H request with a ISO-TP data stream

Since the response to any service 03_H request with more than 2 DTCs in the fault memory will result in a transmission payload of more than 7 bytes, an implementation of the ISO-TP protocol as described in chapter 2.4.6 is required.

The chosen implementation is a state machine (Figure 5-13) which is started when the OBD state machine detects a Service 03_H request. The OBD state machine then periodically calls the ISO-TP state machine until its return value indicates that the transfer is complete. “Complete” in this context also includes “aborted”, since the client has no means of restarting the transfer in case of a problem. This can only be done by the master (in this case, the tester) from the OBD idle state.

To avoid blocking or excessive execution time, no active wait states are included in the state machine. If a resource is not ready or the host requests a transmission interruption, processing is deferred to the next call of the state machine, where the condition is re-evaluated.

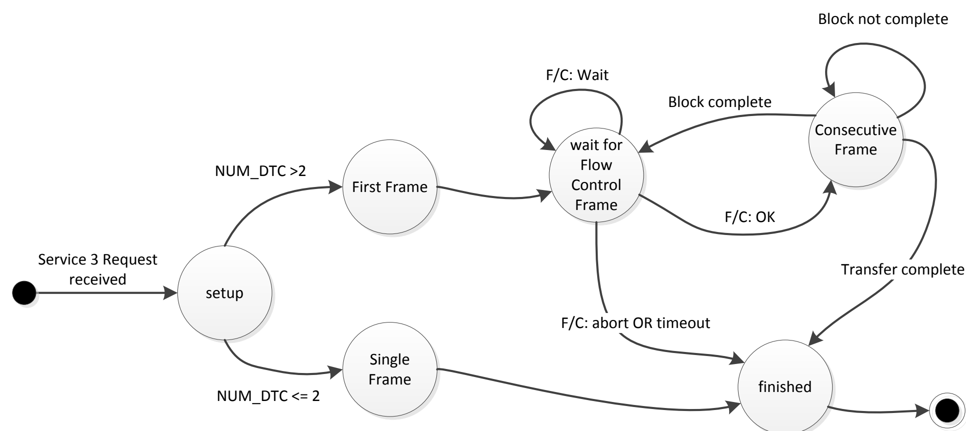


Figure 5-13: ISO-TP state flow

5.2.5 The OBD Vehicle Information Interface

Service 09_H, as described in 2.4.7, provides general Information about the vehicle and control unit connected to the tester.

Per SAE J1979-DA:2011-10 there are 10 standardized INFOTYPES available when using Diagnostics over CAN, summarized in Table 5-1. Evident from this table, only three INFOTYPEs need to be supported in this service 09_H implementation:

- 00_H: list of supported ITIDs
- 04_H: calibration IDs
- $0A_H$: ECU Name

ITID 00_H and $0A_H$ are completely static, meaning they don’t change between different software versions. The CALID, however, should be indicative of the exact software version, meaning that it should be different for each build. In production, this would be e.g.

the build number or the release ID of a certain software build for a specific vehicle. In development, there are often multiple versions in use at the same time and multiple versions may be flashed during a working day. Manually assigning CALIDs to these builds would mean a significant overhead for the developers, as the data has to be manually changed for each build and a database of CALIDs has to be maintained.
Therefore it is reasonable to have the CALID generated automatically so that it is both unique and easily identifiable by the developer.
The way this is achieved in this implementation is to use the time and date of the build. It is easy to cross-reference it to the file date of the firmware files on the PC and is human-readable. To integrate the time and date into the ANSI C predefined macros __DATE__ and __TIME__ are used, which are supported by all standard-compliant C compilers [21] and thus also by the tasking compiler used in the RPEMS toolchain. The resultant CALID is visible in Figure 6-5.

Table 5-1: List of INFOTYPES [21]

ITID	Name	Short	Relevance for Implementation
00_H	Supported INFOTYPES	-	Must be supported
02_H	Vehicle Identification Number	VIN	N/A: vehicles are changing prototypes or engines on testbeds
04_H	Calibration ID	CALID	Relevant: Identifying the software version is an important use-case
06_H	Calibration Verification Numbers	CVN	N/A: anti-tampering measures are not important for this application
08_H	In-use Performance Tracking	-	N/A: Monitors are not supported
$0A_H$	ECU Name	ECUNAME	Relevant: can be useful to identify the RPEMS in a multi-ECU environment
$0B_H$	In-use Performance Tracking	-	N/A: Monitors are not supported
$0D_H$	Engine Serial Number	ESN	N/A: ECU is not tied to a specific engine
$0F_H$	Exhaust Regulation Or Type Approval Number	EROTAN	N/A: not relevant and usually not existent on prototypes
10_H	Protocol Identification	-	N/A: only relevant for UDS implementations

Since all INFOTYPEs (except 00_H) need to be transferred in multiple ISO-TP segments because of their size, an ISO-TP module similar to the one outlined in Figure 5-13 is implemented.

[21] Based on [31]

All code associated with the Vehicle Information Interface can be found in Appendix 5: Listing 7: EOBD_DTC_Interface implementation code

5.2.6 The OBD Debouncer

The modules described in 5.2.1 - 5.2.6 make up the OBD core system, meaning the parts of the software that enable the RPEMS software to communicate with standardized test equipment and which generate and store the data that is provided.
The substantially bigger part of an OBD implementation seeking legislative approval, is the conforming application of the extensive number of monitors and error thresholds to make sure that emission-relevant condition is reliably detected while minimizing the number of false positives.[22]
The implementation described here does have a different goal; nevertheless, a system gating the creation of DTCs and FFs is required to prevent flooding of the fault memory.
A reasonably simple implementation of such a system is the EOBD_debouncer class module. The EOBD_debouncer class gates logic (Boolean) signals that control e.g. the creation of a DTC and takes three parameters:

- The short term (DebST) parameter controls how often the condition has to be detected consecutively, which is used to filter short-term glitches
- The long term (DebLT) parameter controls how often a condition debounced with the DebST parameter has to be detected to enable the output signal. The Boolean output (return value) does not latch; it only generates a single pulse. A continuously appearing condition would therefore, even with DebLT = 1, cause a single pulse (and e.g. a single DTC entry).
- The AutoReArm parameter controls if the debouncer should reset after creating the pulse. If the parameter were logical false it would not generate another pulse.

Figure 5-14 shows an example of EOBD_debouncer behavior over time.
The C source of EOBD_debouncer is listed under Appendix 5:

[22] On systems for the US market, OBD takes up about 30% of the overall time needed for ECU calibration

Listing 1 : EOBD_infotype_interface implementation code

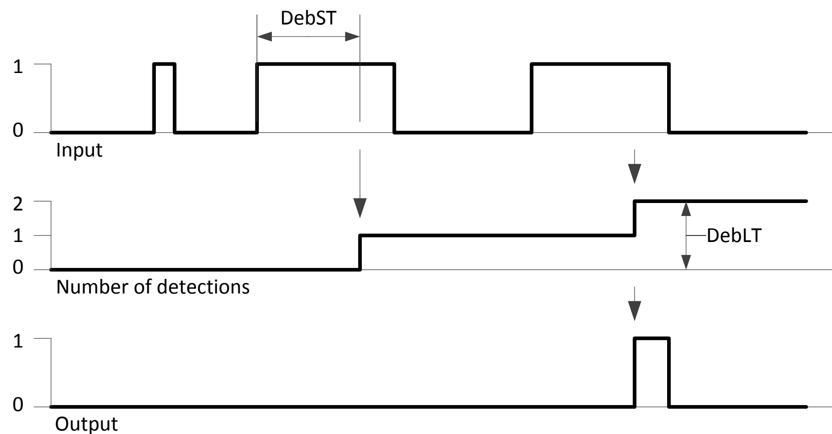


Figure 5-14: Debouncer behaviour

5.3 Diagnostics Implementation example

Figure 5-15 shows a typical way of adding OBD II diagnostic capability to other software modules. The top flowchart shows the conversion of an ADC result (adc0_6) to the throttle pedal position (aped1) in the ADC remapper. KLFWG1 represents characteristic curve, mapping ADC (and this sensor output) voltage to a pedal value between 0% and 100%.
In this case, the valid range for the sensor voltage is 1.0V to 4.0V. If the voltage is outside this range (comparison in the lower left) once (debLT =1) for at least 100ms (debST = 100cycles at 1ms intervals) the debouncer calls “StoreDTC” from the imported EOBD_DTC_interface class with the DTC code P0120 (Throttle/Pedal Position Sensor/Switch "A" Circuit Malfunction). A freeze frame is not stored and the MIL is not lit (store_FF and enables_MIL are false).

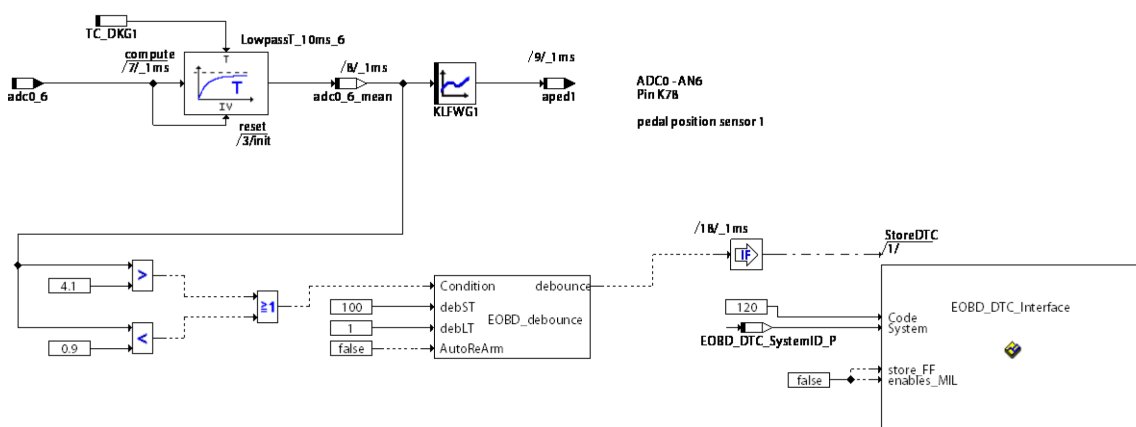


Figure 5-15: Exemplaric implementation of trottle pedal diagnostics

5.4 Architecture of the OBD subsystem

Figure 5-16 shows how the OBD software modules interface with each other and the rest of the RPEMS software. The run_NNms functions are attached to the respective cyclic tasks in the task scheduler. The modules shaded in grey are existing RPEMS software elements the OBD software interacts with.

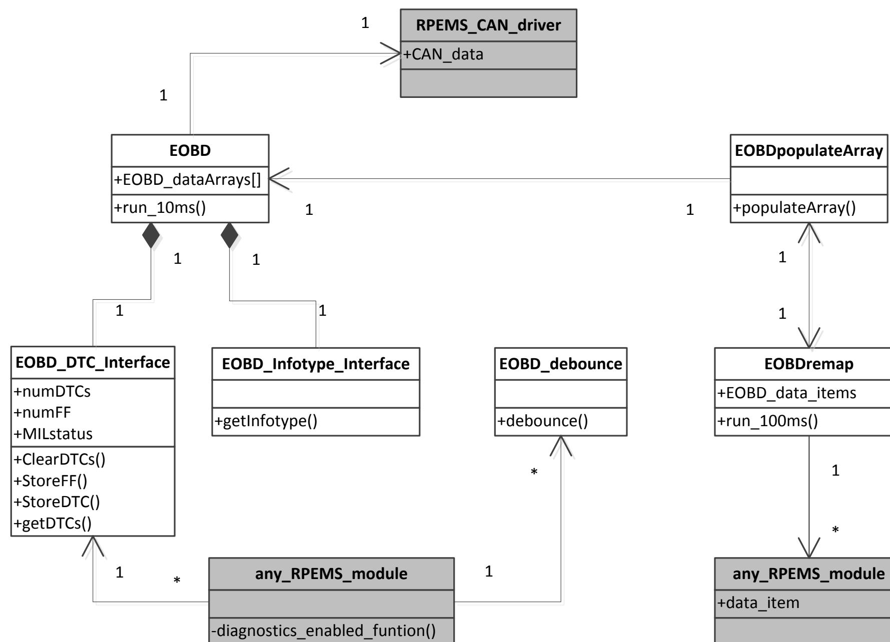


Figure 5-16: OBD software structure

The EOBD module represents the OBD state machine from 5.2.2, which also integrates the DTC and Infotype interface. To gain access to the CAN bus, the EOBD module imports the relevant data elements from the RPEMS' CAN software stack.
To provide recent data to the OBD system, EOBDremap converts supported variables from other software modules to the format required by OBD every 100ms. After the conversion, it calls EOBDpopulateArray, which has access to the main OBD arrays and copies the re-mappers' converted data items there.
Any RPEMS software that wants to use the diagnostic system just imports EOBD_DTC_Interface (and optionally EOBD_debounce) to use their public functions to store DTCs and freeze frames.

5.4.1 Timing

In the OBD II standards, the time $P2_{CAN}$ is defined as the maximum response time to any request. The response time is defined as the time between reception of an OBD II request and the response in form of an ISO_TP single- or first frame. $P2_{CAN,max}$ is 50 milliseconds,

which means that every tester should wait for an response at least 50ms while no ECU should take longer than 50ms to respond.[22]
Since data is passed between multiple processes running in the 10ms grid, a few measures have to be taken to make sure this deadline is met. The processes involved in producing the response (in the order data passes through them during an OBD request/response) are listed in Table 5-2.

process in module::process notation	task	Description
MultiCAN::receiveNode3_10ms	_10ms	get CAN request data from MOb
CANremap::receive_10ms	_10ms	remap CAN request data and perform flow control
EOBD::EOBD_handler_10ms	_10ms	parse request and craft response (x2/x3)[23]
CANremap::transmit_10ms	_10ms	remap response data and perform flow control
MultiCAN::transmitNode3_10ms	_10ms	load response data in CAN MOb

Table 5-2: processes needed to answer an OBD request

The time needed for the CAN data to be moved between the bus and the corresponding MObs can be considered as nearly instant (<0,5ms at 500kBit/s), so the response time depends mainly on the number of executions of the _10ms task necessary to complete the operation.
Figure 5-17 shows the best and worst case scenario of how the order of execution of processes inside a task can affect the total time to completion of an OBD II request. The grayed-out blocks represent processes which cannot do useful work during the respective execution cycle.
In Figure 5-17 A, the processes are optimally arranged inside the task; in the first and third run, multiple processing steps can be executed because the interdependent processes were called in the correct order by the task scheduler. Figure 5-17 B is the worst-case scenario – each processing step has to be executed in a subsequent call of the _10ms task. No speedup is possible since no two processes are executed in the correct order and so the $P2_{CAN}$ deadline is missed by at least 20ms.

[23] This step takes more than one cycle. During the first execution the OBD state machine will change its state depending on the request. For single-frame responses, the response is created during the second execution. When a multi-frame response is required, one additional cycle is required to setup the ISO_TP state machine.

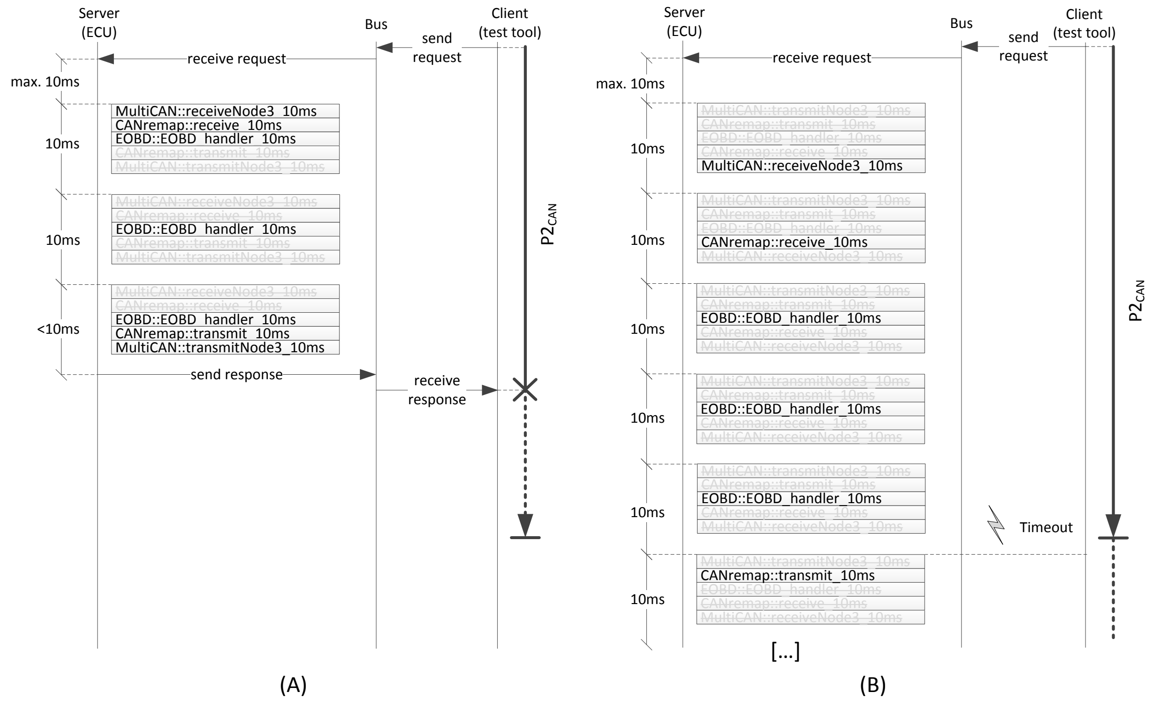


Figure 5-17: OBD response time

Chapter 6

Testing, Results and Outlook

6.1 Test Setup

The setup which was used during development and testing is laid out in Figure 6-1; Details are listed in Table 6-1 and Table 6-2. For the use with standard OBD testers an adapter harness had to be manufactured to provide power to the tester and to allow the connection of a third node to the CAN bus connected to CAN node 3 on the RPEMS.

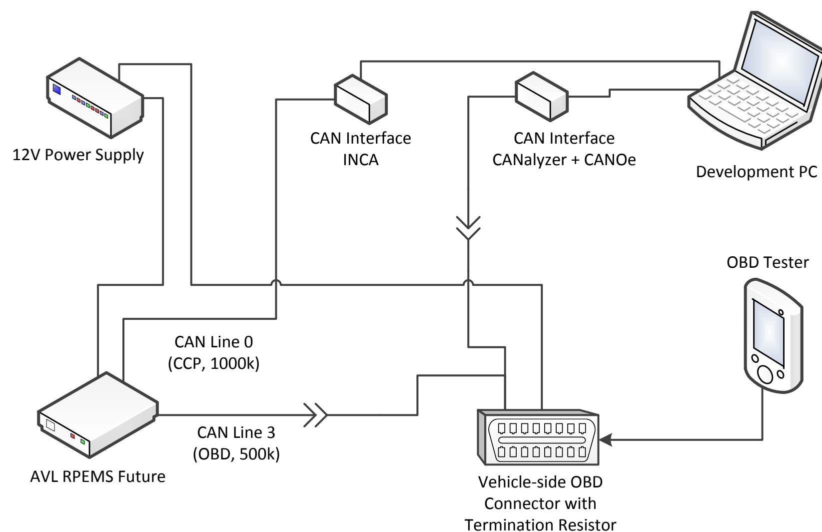


Figure 6-1: Test Hardware Setup

Table 6-1: Test Hardware

Item	Description
12V Power Supply	230V mains voltage to 12VDC / 3A regulated power supply
AVL RPEMS Future	RPEMS Future, HW Version 1.3, VCU Configuration[24], modified to have a termination resistor on CAN interface 3
CAN Interface INCA	ETAS ES580 PCMCIA to CAN Interface (identical to vector CANCardXL), with Type 251 Transceiver Cable (82C251Tranceiver Chip)
CAN Interface CANalyzer + CANoe	vector CANcaseXL or vector VN1610 (depending on application due to licensing limitations)
OBD Tester 1	Autel Maxiscan MS509 OBD II/EOBD diagnostic tester with built-in DTC database
OBD Tester 2	Generic ELM327-based OBD-to-Bluetooth Interface

[24] RPEMS in VCU configuration do not have their high-power and high-voltage output drivers populated, as they do not have to drive injectors and ignition coils

Table 6-2: Test Software

Vendor	Product Name	Functions
ETAS	ASCET 5.2.2	Software Specification
ETAS	INCA 7.0.0/14	Firmware flashing Parameterization Runtime Variable Monitoring
Vector	CANalyzer 7.6.27	Interactive sending and receiving of CAN data
Vector	CANoe 8.1	Run automated test cases Software OBD Tester
Vector	CANoe.DiVa 3.1 SP1	Generate automated tests for OBD II Implementations

6.2 Early Testing

ASCET 5.2 only supports the software development cycle up to the build process, there are no testing or debugging facilities integrated.

Limited debugging can be done by using INCA; after flashing a new software file on the ECU, INCA can establish a CCP connection and use the A2L[25] file generated by ASCET.

This provides access to the values of all data items marked for calibration in ASCET. If the value in question has been marked as "Parameter", it can also be changed, which serves as a limited method to provide stimulus. The limitations of this method are:

- There is no way to slow down, pause, or single-step execution
- Since the CCP protocol runs as a regular process on the RPEMS, the highest possible resolution is one execution of a given process, intermediate values cannot be viewed
- Due to an unconfirmed bug in the handling of parameters, they cannot be used with class modules that have more than one instance
- Since the CCP protocol is implemented in software on the ECU, it cannot be used if the CAN modules or the initialization is not working correctly or if another software module hangs.

The first module to be implemented was EOBD_remap to test the value conversion using formulas, which could easily be confirmed by having the values displayed in INCA.

[25] An ASAP2 description file (also called A2L) contains all information on the relevant data objects in the ECU such as characteristics (parameters, characteristic curves and maps), real and virtual measurement variables and variant dependencies. Information is needed for each of these objects, such as memory address, storage structure, data type and conversion rules for converting them into physical units.[32]

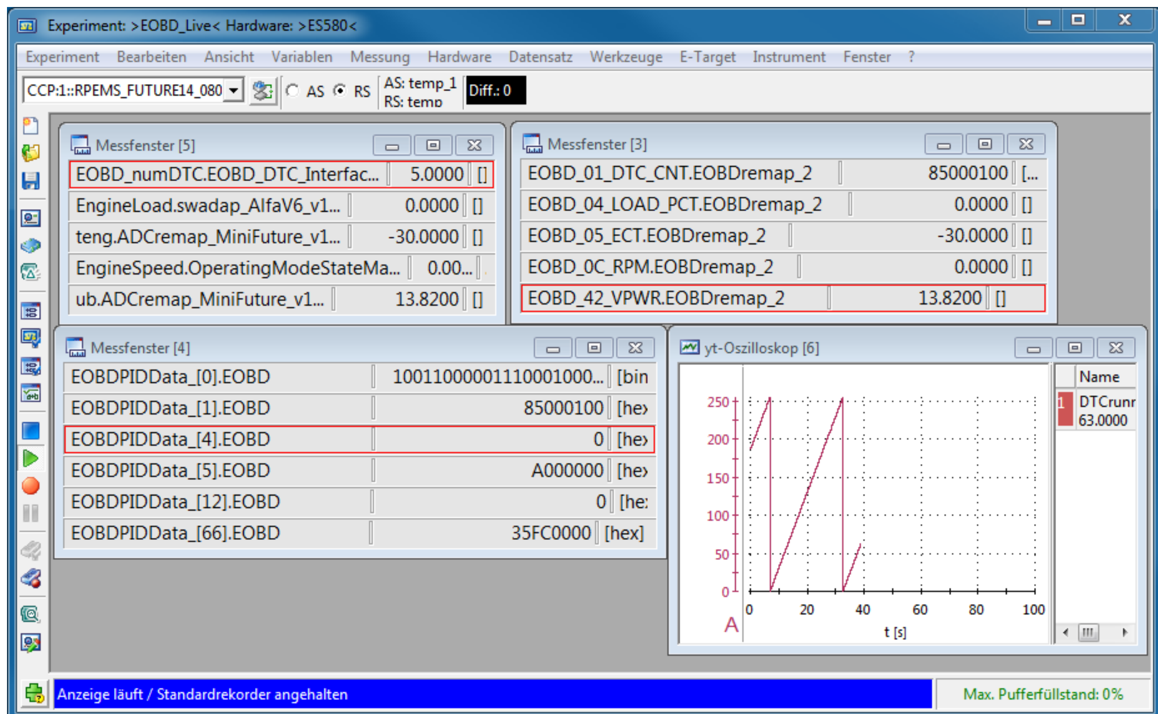


Figure 6-2: INCA I Experiment showing data from the OBD remapper

The next milestones were the modified CAN software and the foundations of the OBD state machine. The first real challenge was the persistent storage interface, because of its implementation in C. Variables declared in C code are not added to the A2L file, therefore helper variables had to be introduced to monitor them. Also, simple bugs like boundary violations or null-pointer exceptions were hard to debug because they crashed the ECU which disabled CCP.

6.3 Communication tests

Early communication tests were carried out by sending hand-crafted CAN packets mimicking an OBD tester using Vector CANalyzer. Once the OBD software was able to send meaningful responses, most tests were carried out using the Autel standalone scan tool, which supports most functions implemented. Only the “ECU Name” Infotype and multiple freeze frames are not supported; these functions were tested using the App Torque on an LG Nexus 4 Android smartphone and the ELM327 Bluetooth interface. The adapter harness, as mentioned above, enabled the connection of CANalyzer as third node on the bus to trace the communication between tester and ECU and helped working out, at which point it went wrong.

Figure 6-3 shows an unsuccessful connection attempt of the tester (repeat “02 01 0C …“ messages) in the trace window and faux ECU responses (ID 7E8) in the generator window. Figure 6-4 shows the Autel MaxiScan displaying the DTC code P0666 after successfully reading four DTCs (top right display corner) from the persistent storage interface.

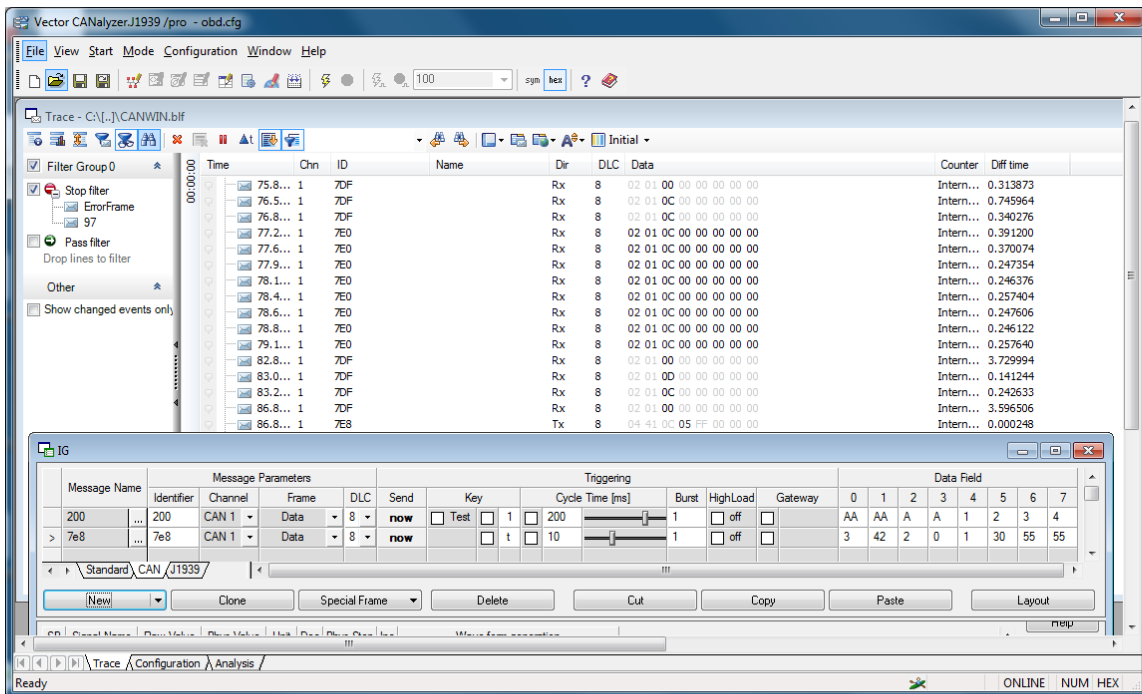


Figure 6-3: CANalyzer used for communication tests

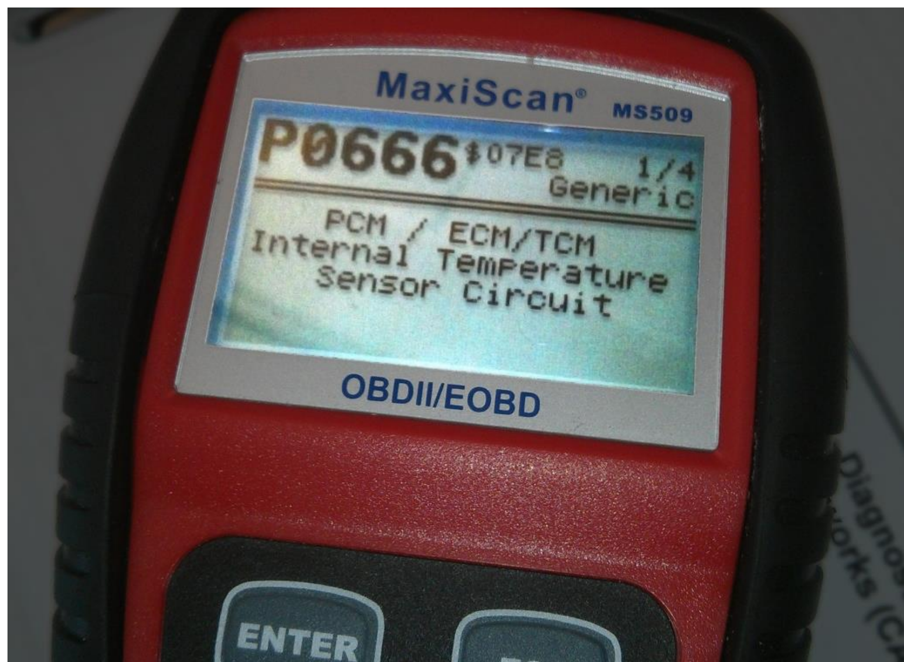


Figure 6-4: Successful DTC readout using the Autel scan tool

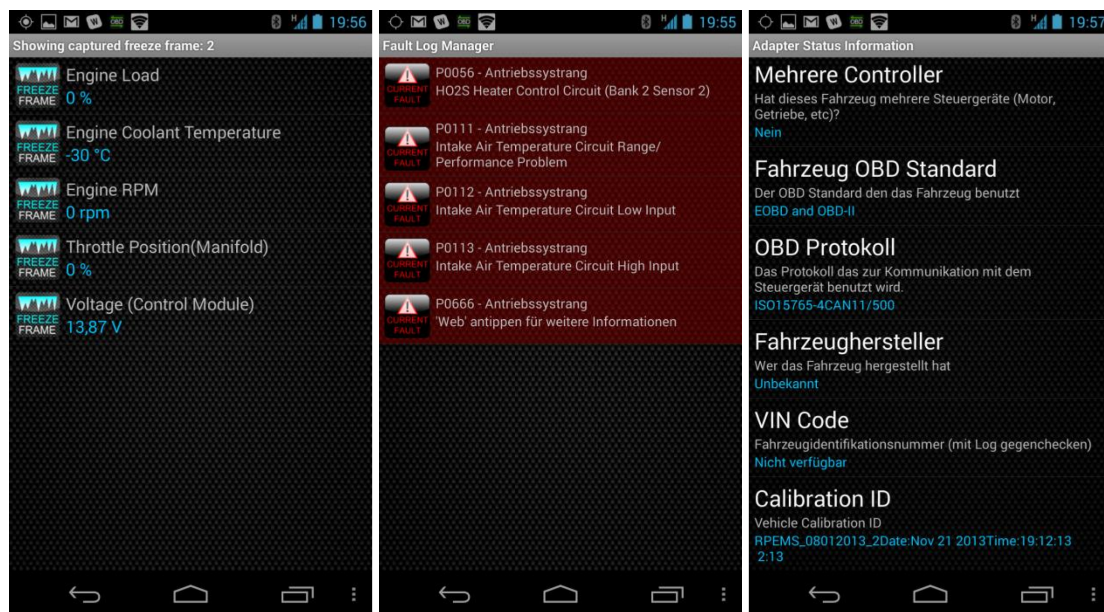


Figure 6-5:Torque App displaying data from OBD II services 02_H 03_H and 09_H

6.4 Systematic testing

While working data displays on the testers show that the implementation works in principle, it does not show that it is really compliant to the standards. The testers only represent “good case” tests, since they should always produce valid requests and are usually, aiding robustness, very tolerant regarding violations by the ECU’s implementation.

Also, even all possible valid and supported combinations of SID, PID, etc… would present a large number of test cases, taking a lot of time to go through.

A more systematic approach is to use a tool that automatically generates test cases, runs them and analyzes the responses.

A software tool that does this is CANoe.DiVa[26], an extension to Vector’s CANoe ECU software development environment designed to test implementations of diagnostic services.

“DiVa is a CANoe extension for automated testing of diagnostic software implementations in ECUs. Reproducible test cases are generated based on an ECU diagnostic description.” [23]

DiVa has a built-in description of the standard services based on the standards mentioned in chapter 2.4 and generated over 400 test cases applicable to this implementation (see Appendix 2: CANoe.DiVa Test Specification).

The first test run indeed found a lot of implementation problems which had left the testers unfazed; only 48% of the tests resulted in a pass.

Most of the errors were due to a wrong ISO-TP header format and a couple of builds later, these kinks were ironed out and CANoe reported a pass rate of 100% (see Appendix 3: CANoe.DiVa Test Results), confirming the compliance of the protocol implementation.

[26] DiVa = Diagnostics Validation

6.5 Conclusion and Outlook

The OBD software module has been implemented successfully and has proofed its functioning in several test environments.

Once integrated, with little configuration, it allows viewing live data, e.g. from sensors. To take advantage of all provided functionality some work has to be invested by the individual modules' to define diagnostic thresholds and conditions to store DTCs and Freeze Frames.

Not all OBD services were supported, mostly because the effort required for their calibration would exceed their usefulness (Monitors and Tests). A possible area of future work would be the inclusion of a multi-tiered fault memory distinguishing between sporadic and confirmed faults (service 07_H).

Another goal for optimization could be the further reduction of the memory footprint of the OBD II module; the current version is optimized for ease of use and readability, which wastes some memory in the layout of the main data and configuration arrays.

During the writing of this thesis, a second project has been started to integrate an SD-card interface into the RPEMS hardware, so that the unit can store extended logging data on board, which should extend the diagnostic abilities of the system even further. A combination with the OBD module would be beneficial, e.g. for marking the acquired data on the SD card when OBD events happen.

Bibliography

[1] California Enviromental Protection Agency Air Resources Board, "Key Events in the History of Air Quality in California," 2012. [Online]. Available: http://www.arb.ca.gov/html/brochure/history.htm. [Accessed: 05-Jul-2013].

[2] W. Grant, *Autos, smog, and pollution control: the politics of air quality management in California*. Edward Elgar Publishing Company, 1995.

[3] J. E. Krier and E. Ursin, *Pollution and Policy: A Case Essay on California and Federal Experience with Motor Vehicle Air Pollution, 1940-1975*. University of California Press, 1977.

[4] T. Schneider, *Air Pollution in the 21st Century: Priority Issues and Policy*. Elsevier, 1999.

[5] K. Beiter, C. Rätz, and O. Garnatz, "Gesetzliche On-Board-Diagnose und ODX," in *Diagnose in mechatronischen Fahrzeugsystemen III : neue Verfahren für Test, Prüfung und Diagnose von E/E-Systemen im Kfz*, 2010, pp. 44–56.

[6] European Parliament, *Directive 98/69/EC of the European Parliament and of the Council of 13 October 1998 relating to measures to be taken against air pollution by emissions from motor vehicles and amending Council Directive 70/220/EEC*. EU: EUR-Lex, 1998.

[7] D.-I. F. Albrecht, D.-I. M. Krauß, and D.-I. G. Meder, "Der neue Sechszylindermotor mit 3 l Hubraum von BMW für die ULEV-Abgasgesetzgebung und OBD," *MTZ - Mot. Zeitschrift*, vol. 61, no. 11, pp. 750–757, Nov. 2000.

[8] European Parliament, *Regulation (EC) No 715/2007 of the European Parliament and of the Council of 20 June 2007 on type approval of motor vehicles with respect to emissions from light passenger and commercial vehicles (Euro 5 and Euro 6) and on access to vehicle repair and mai*. EU, 2007.

[9] W. Zimmermann and R. Schmidgall, *Bussysteme in der Fahrzeugtechnik: Protokolle, Standards und Softwarearchitektur*, vol. 2010. Vieweg+Teubner Verlag, 2010.

[10] ISO International Organization for Standardization, "ISO 15031-4: Communication between vehicle and external equipment for emissions-related diagnostics — Part 4: External test equipment," 2005.

[11] Kvaser Inc., "Kvaser Memorator Light HS." [Online]. Available: http://www.kvaser.com/index.php?option=com_php&Itemid=259&eaninput=7330130005136&lang=en&product=Kvaser Memorator Light HS.

[12] Vector Informatik, "Vector GL1000 Series Logger." [Online]. Available: http://vector.com/vi_gllogger_de.html#!../vi_gllogger_gl1000_iframe_de.html. [Accessed: 11-Dec-2013].

[13] S. McConnell, *Software Estimation: Demystifying the Black Art: Demystifying the Black Art*, vol. 2009. O'Reilly Media, Inc., 2009.

[14] Phytec Europe, "phyCORE-TC1796 / TC1797 / TC1793 product overview." [Online]. Available: http://www.phytec.eu/europe/products/modules-overview/cutting-edge-soms/product-details/p/phycore-tc1796tc1797.html. [Accessed: 20-Aug-2013].

[15] E. W. Dijkstra, "Cooperating sequential processes," in in *The origin of concurrent programming*, New York: Springer-Verlag New York, Inc., 2002, pp. 65–138.

[16] R. C. Seacord, *Secure Coding in C and C++ (Google eBook)*. Pearson Education, 2005, p. 368.

[17] P. Mandl, *Grundkurs Betriebssysteme: Architekturen, Betriebsmittelverwaltung, Synchronisation, Prozesskommunikation (Google eBook)*. Springer DE, 2009.

[18] Q. Li and C. Yao, *Real-Time Concepts for Embedded Systems*. 2003.

[19] W. Müller, "Entwicklung von Klopfregelfunktionen für das RPEMS NG Steuergerät," 2009.

[20] ETAS GmbH, *Ascet v5.2 Manual*. 2007.

[21] ISO/IEC, *ISO/IEC 9899:201x Programming languages — C*. 2011.

[22] ISO International Organization for Standardization, "ISO 15031-5: Communication between vehicle and external equipment for emissions-related diagnostics — Part 5: Emissions-related diagnostic services," 2011.

[23] Vector Informatik, "CANoe.DiVa - Automated Testing of the Diagnostic Protocol in ECUs." [Online]. Available: http://vector.com/vi_canoediva_en.html. [Accessed: 27-Nov-2013].

[24] ISO International Organization for Standardization, "ISO 15031-3: Communication between vehicle and external equipment for emissions-related diagnostics — Part 3: Diagnostic connector and related electrical circuits, specification and use," 2004.

[25] ISO International Organization for Standardization, "ISO 15765-2: Diagnostics on Controller Area Networks (CAN) — Part 3: Implementation of unified diagnostic services (UDS on CAN)," 2004.

[26] ISO International Organization for Standardization, "ISO 11898-2: Controller area network (CAN) — Part 2: High-speed medium access unit," 2003.

[27] ISO International Organization for Standardization, "ISO 11898-1: Data link layer and physical signalling; Road vehicles — Controller area network (CAN)," 2003.

[28] NXP Semiconductor, "PCA82C251 CAN transceiver for 24 V systems," no. August. 2011.

[29] User: Fröstel, “File:CAN telegramm 2.0A.svg - Wikimedia Commons.” [Online]. Available: http://commons.wikimedia.org/wiki/File:CAN_telegramm_2.0A.svg.

[30] ETAS GmbH, *Ascet v5.2 Reference Guide*. 2007.

[31] SAE Society for Automotive Engineers, “SAE J1979-DA,” 2011.

[32] Vector Informatik, “Description Files for Internal ECU Parameters.” [Online]. Available: http://vector.com/vi_datadescription_ecu1_en.html. [Accessed: 10-Nov-2013].

Appendix

Appendix 1: List of Abbreviations ... 71

Appendix 2: CANoe.DiVa Test Specification ... 72

Appendix 3: CANoe.DiVa Test Results ... 73

Appendix 4: Software Module Block Diagrams ... 74

Appendix 5: Source Code ... 83

Appendix 1: List of Abbreviations

ADC	Analog to Digital Converter
ASCET	Advanced Simulation and Control Engineering Tool
CALID	Calibration ID
CAN	Controller Area Network
CARB	California Air Resources Board
CCP	CAN Calibration Protocol
CSMA/CR	Carrier Sense Multiple Access / Collision Resolution
DAVE	Digital Application Virtual Engineer
DTC	Diagnostic Trouble Code
ECU	Engine Control Unit
EEPROM	Electrically Erasable Programmable ROM
EMI	Electromagnetic Interference
EMS	Engine Management System
EOBD	European On Board Diagnostics
ESDL	Embedded Systems Description Language
FF	Freeze Frame
GDI	Gasoline Direct Injection
ID	Identifier
INCA	Integrated Calibration and Acquisition System
ITID	InfoType ID
KWP	Key Word Protocol
MCU	Micro Controller Unit
MIL	Malfunction Indicator Light
MOb	Message Object
MY	Model Year
OBD	On Board Diagnostics
OSI	Open Systems Interconnect
PID	Parameter ID
RPEMS	Rapid Prototyping Engine Management System
SID	Service ID
SoM	System on Module
SPI	Serial Peripheral Interface a.k.a. Four-Wire-Bus
TID	Test ID
TP	Transport Protocol

Appendix 2: CANoe.DiVa Test Specification

DiVa Test Specification

Table of contents

ECU Specification

Tests (401 generated)

1 Powertrain Diagnostic Data (139 tests)

1.1 Read Supported PIDs (1 test)

1.2 Read PIDs (138 tests)

2 Monitored Systems (89 tests)

2.1 Read Supported MIDs (2 tests)

2.2 Read MIDs (87 tests)

3 System, Test or Component (10 tests)

3.1 Read Supported TIDs (1 test)

3.2 Read TIDs (9 tests)

4 Vehicle Information (20 tests)

4.1 Read Supported Info Types (1 test)

4.2 Read Info Types (19 tests)

5 Fault Memory (143 tests)

5.1 Emission-Related Diagnostic Trouble Codes (1 test)

5.1.1 Request Emission-Related DTCs (1 test)

5.2 Freeze Frame Data (139 tests)

5.2.1 Read Supported PIDs (1 test)

5.2.2 Read Freeze Frame Data (138 tests)

5.3 Emission-Related Diagnostic Trouble Codes Detected During Current or Last Completed Driving Cycle (1 test)

5.3.1 Request Emission-Related DTCs (1 test)

5.4 Request Emission-Related Diagnostic Trouble Codes with Permanent Status (1 test)

5.4.1 Request Emission-Related DTCs (1 test)

5.5 Clear/Reset Emission-Related Diagnostic Information (1 test)

5.5.1 Clear Emission-Related DTCs (1 test)

5.6 Active Fault Memory test (0 tests)

5.6.1 Test Emission-Related DTCs (0 tests)

6 Transport Layer (0 tests)

ECU Specification

ECUInfo [OBD]	
File	
Version	1.20
Name	OBD
Variant	COMMON_DIAGNOSTICS

Appendix 3: CANoe.DiVa Test Results

Report CANoe.DiVa TestModule

Test passed

General Test Information

Test Engineer

Windows Login Name:	Luke

Test Setup

Version:	CANoe 8.1.60 (SP2)
Configuration:	C:\Users\Public\Documents\Vector\CANoe\8.1\CANoe Demos\OBD.cfg
Configuration Comment:	
Test Module Name:	RPEMS2
Test Module File:	C:\Users\Luke\Dropbox\Diplomarbeit\DiVa\RPEMS2.vxt
Last modification of Test Module File:	2013-11-21, 18:24:22
Test Module Library (CAPL):	C:\Users\Luke\Dropbox\Diplomarbeit\DiVa\RPEMS2.callback.can
Test Module Library (CAPL):	C:\Users\Luke\Dropbox\Diplomarbeit\DiVa\RPEMS2.001.can
Windows Computer Name:	LUKE-HP
Nodelayer Module osek_tp:	OSEK_TP (VC10, Version 5.20.51, Build 51), C:\Program Files (x86)\ Vector CANoe 8.1\Exec32\osek_tp.dll
Nodelayer Module DiVa:	DiVaTestExtension, C:\Program Files (x86)\Vector CANoe 8.1\Exec32\DiVa\DiVa.dll
Diagnostic Specification:	
DiVa Project:	C:/Users/Luke/Dropbox/Diplomarbeit/DiVa/RPEMS2.diva
DiVa Configuration:	Config
ECU Name:	OBD
ECU Variant:	COMMON_DIAGNOSTICS
Specification Version:	1.20
TestModuleGenerationTime:	2013-11-21T17:24:21.048

Start Timings

P2 Time:	150ms
P2* Time:	2000ms
S3 Time:	5000ms
EcuTimings:	This are only the start Timings. P2 and P2* will change as specified by the response of session control services.

P2 Time Metrics

P2Time Average:	37ms
P2Time Min.:	30ms
P2Time Max.:	50ms

Test Overview

Test begin: 2013-11-21 19:21:33 (logging timestamp 3359.266915)
Test end: 2013-11-21 19:23:46 (logging timestamp 3492.568527)

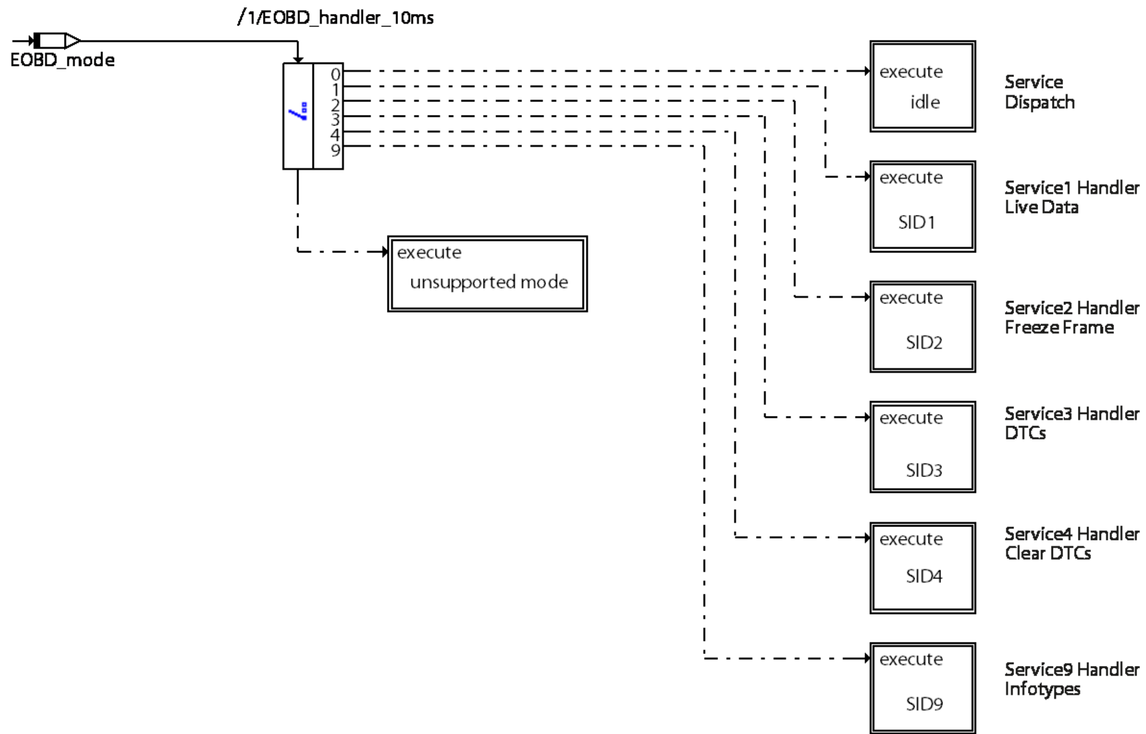
Statistics

Overall number of test cases	401	
Executed test cases	401	100% of all test cases
Not executed test cases	0	0% of all test cases
Test cases passed	401	100% of executed test cases
Test cases with warning	0	0% of executed test cases
Test cases failed	0	0% of executed test cases

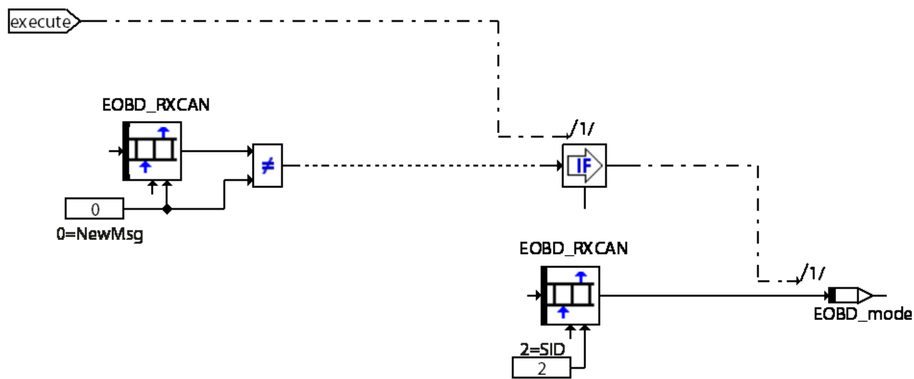
Appendix 4: Software Module Block Diagrams

List of Block Diagrams

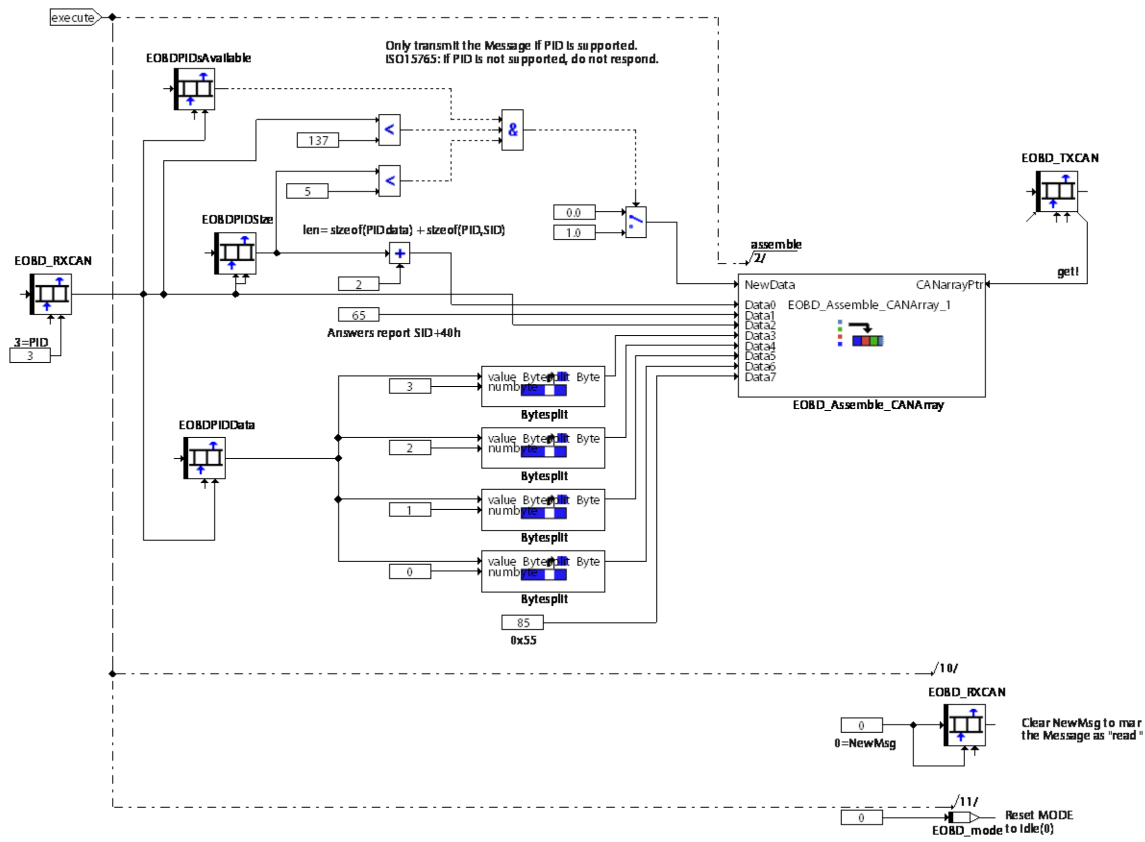
Block Diagram 1: EOBD state machine main Diagram 75
Block Diagram 2: EOBD state machine idle state subdiagram 75
Block Diagram 3: EOBD state machine SID 1 subdiagram 76
Block Diagram 4: EOBD state machine SID 2 subdiagram 76
Block Diagram 5: EOBD state machine SID 3 subdiagram 77
Block Diagram 6: EOBD state machine SID 4 subdiagram 77
Block Diagram 7: EOBD state machine SID 9 subdiagram 78
Block Diagram 8: EOBD state machine invalid SID subdiagram 78
Block Diagram 9: EOBD_DTC_Interface test program 79
Block Diagram 10 : EOBD_remapper main diagram 80
Block Diagram 11: EOBD_remapper EOBD_01_DTC_CNT subdiagram 81
Block Diagram 12: EOBD_remapper init subdiagram 81
Block Diagram 13: EOBD_debouncer 82



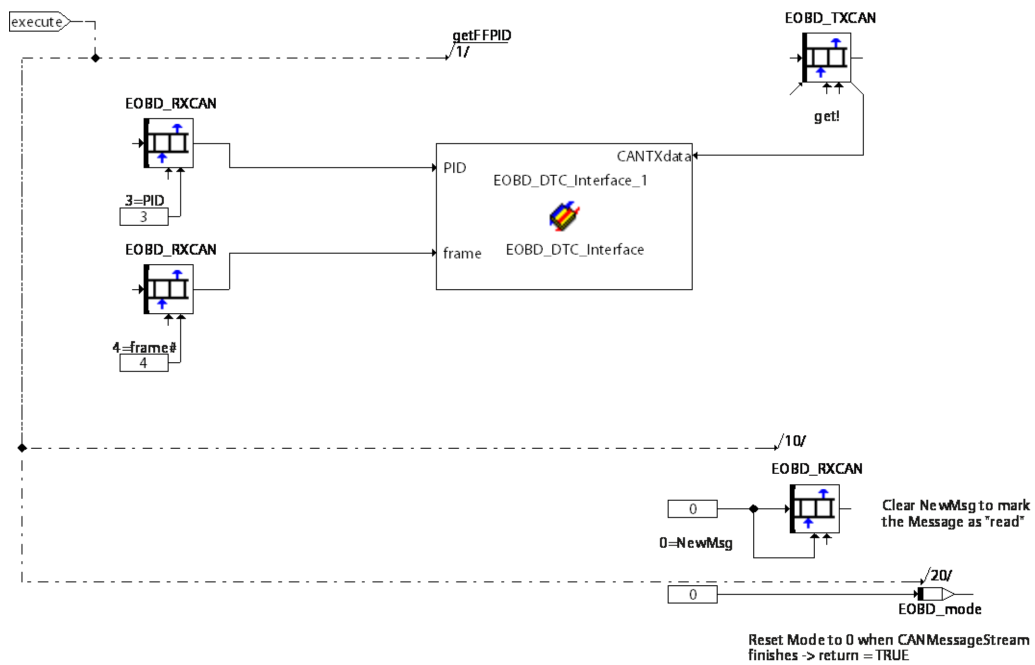
Block Diagram 1: EOBD state machine main Diagram



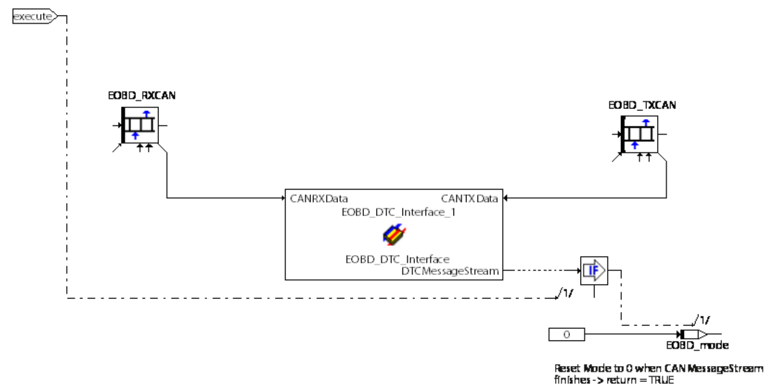
Block Diagram 2: EOBD state machine idle state subdiagram



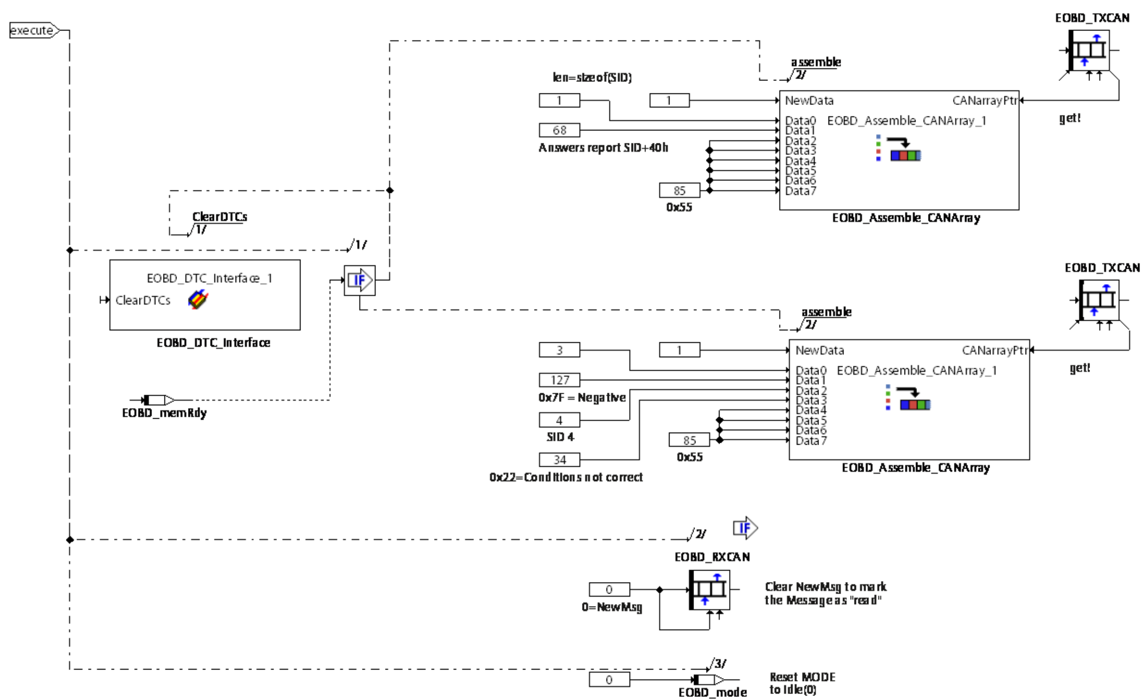
Block Diagram 3: EOBD state machine SID 1 subdiagram



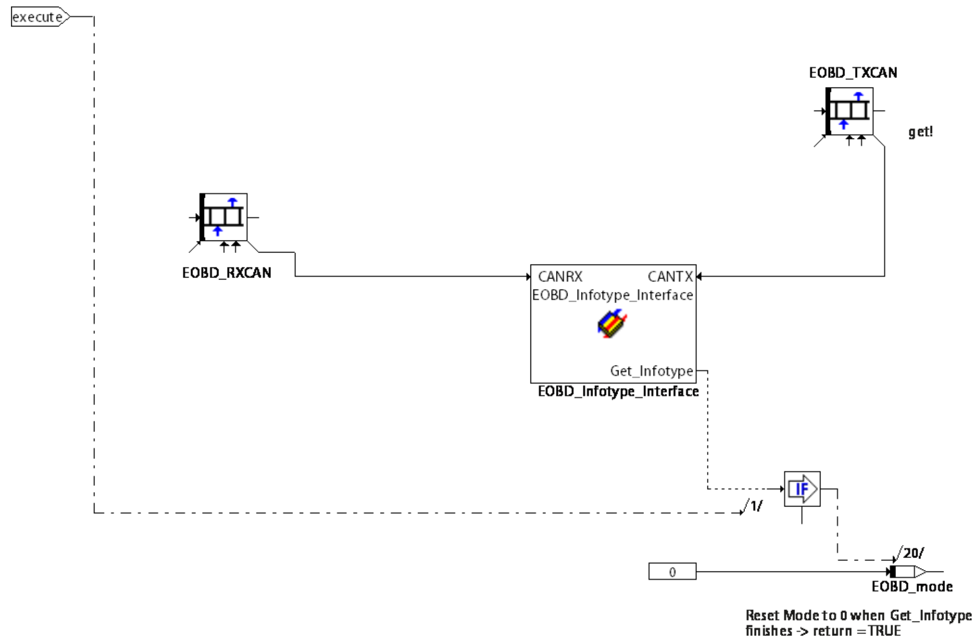
Block Diagram 4: EOBD state machine SID 2 subdiagram



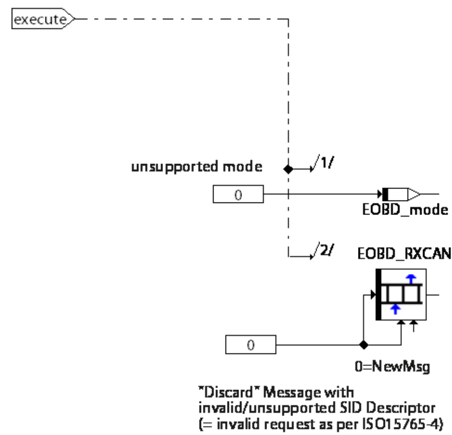
Block Diagram 5: EOBD state machine SID 3 subdiagram



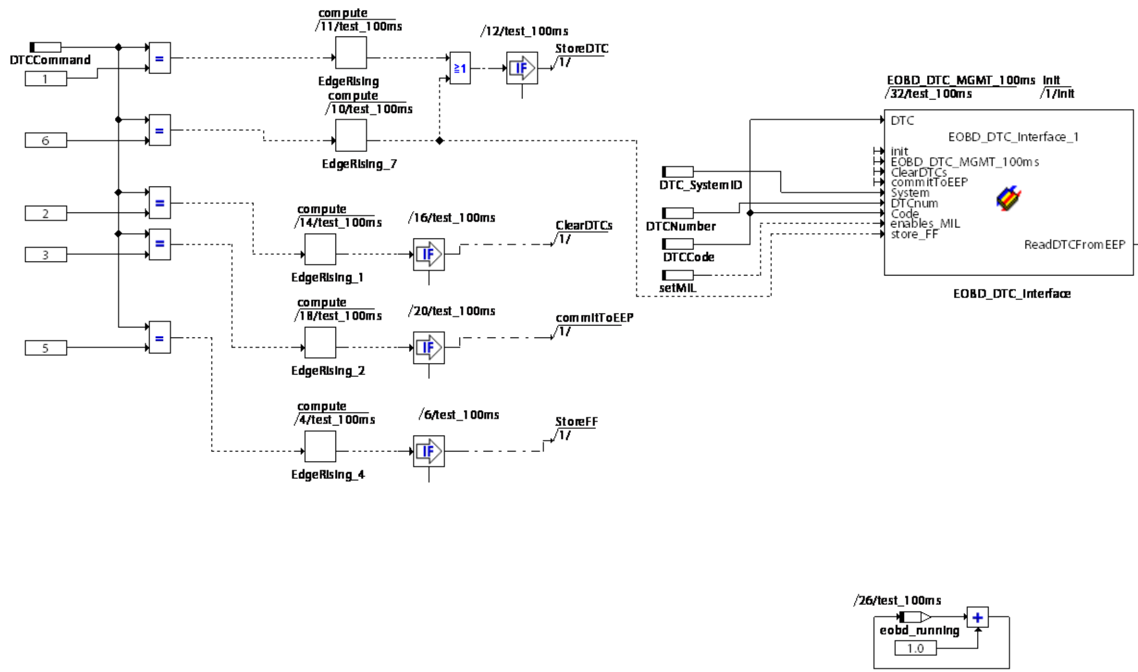
Block Diagram 6: EOBD state machine SID 4 subdiagram



Block Diagram 7: EOBD state machine SID 9 subdiagram



Block Diagram 8: EOBD state machine invalid SID subdiagram



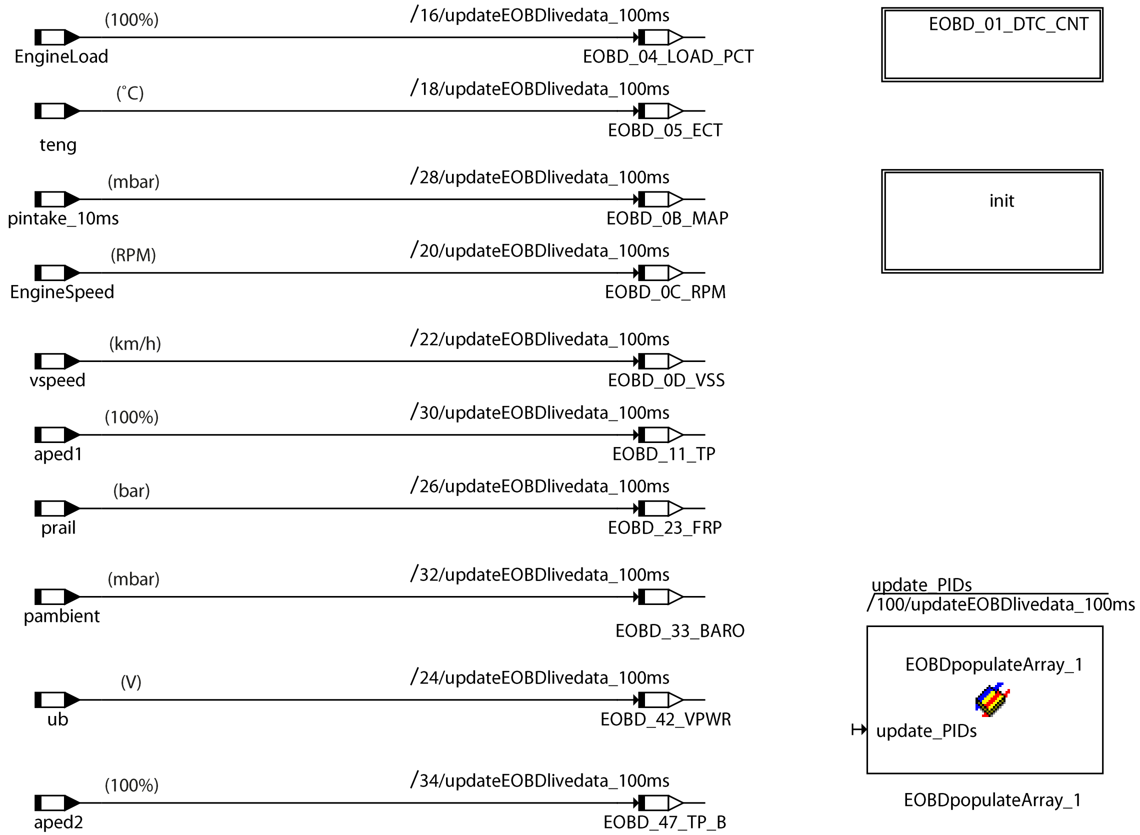
Block Diagram 9: EOBD_DTC_Interface test program

EOBDPIDData contains the Live Data items, indexed by OBD PID#
EOBDPIDsAvailable contains a bitfield that encodes the PIDs available on this ECU.
(see data_update)

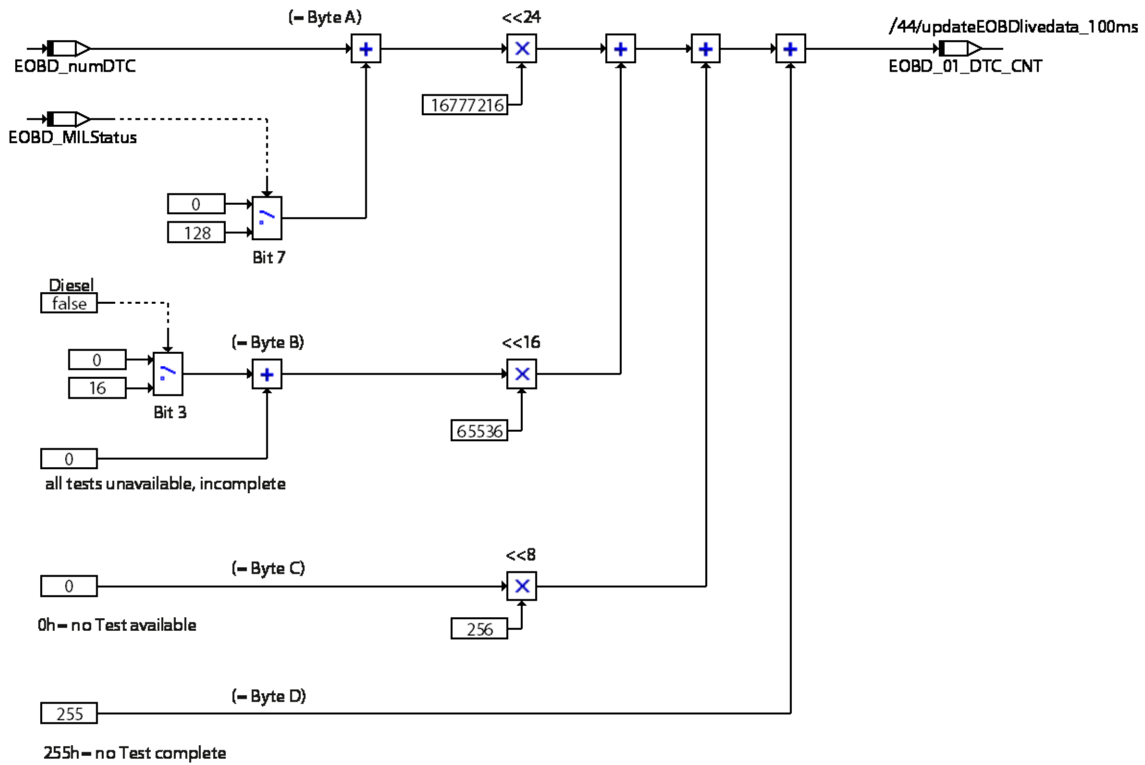
So for every Data item set, the corresponding Capabilities bit has to be set, too, so that the ECU can report the availability to the tester.
(online) edit EOBDPIDsAvailable to manually activate/deactive available PIDs on the OBD Interface

The EOBD_xx_yyyy Variables are used for conversion.
Each has a formula attached to it that converts the physical data in the format used by OBD

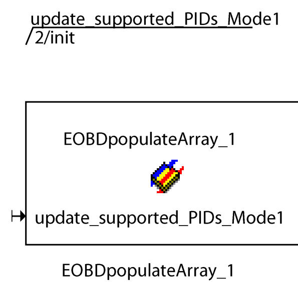
This Module has to be called in pair with
updateEOBDlivedata::updateEOBDpopulateArray



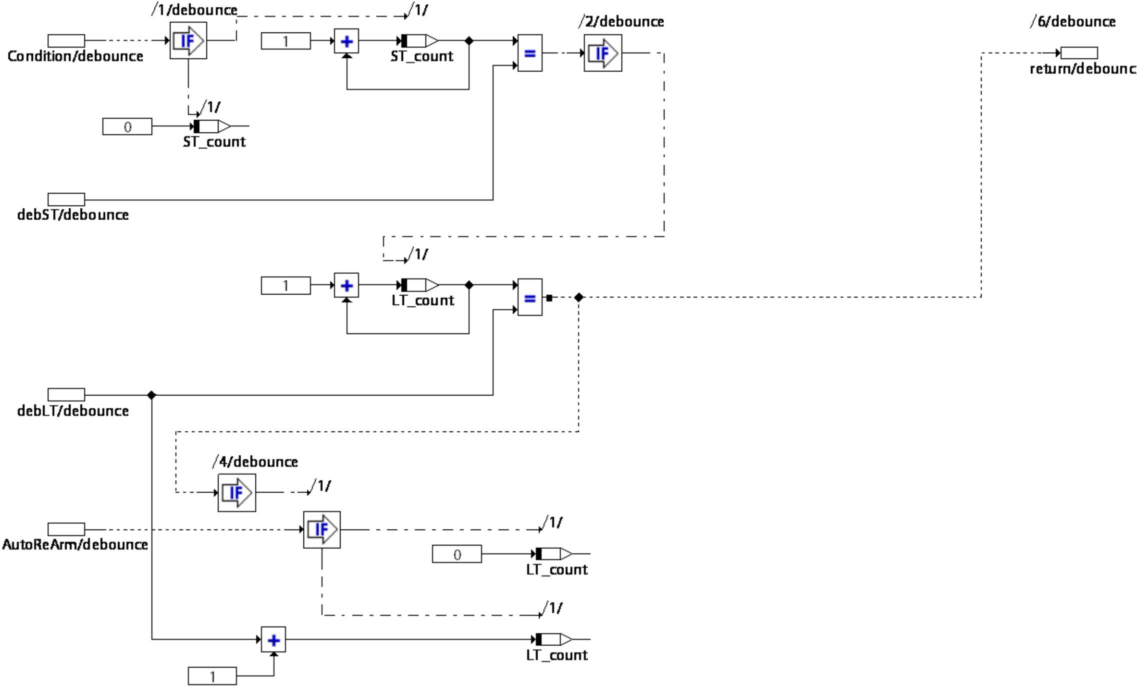
Block Diagram 10 : EOBD_remapper main diagram



Block Diagram 11: EOBD_remapper EOBD_01_DTC_CNT subdiagram



Block Diagram 12: EOBD_remapper init subdiagram



Block Diagram 13: EOBD_debouncer

Appendix 5: Source Code

List of Listings

Listing 1 : EOBD_infotype_interface implementation code 84

Listing 2: EOBD_infotype_interface header code 89

Listing 3: EOBD_remap::update_supported_PIDs_Mode1() implementation code 89

Listing 4: EOBDpopulateArray::data_update_100ms() header code 90

Listing 5: EOBDpopulateArray::data_update_100ms implementation Code 90

Listing 6: EOBD_DTC_Interface header code 91

Listing 7: EOBD_DTC_Interface implementation code 92

Listing 8: EOBD_bytesplit implementation code 109

Listing 9:EOBD_assemble_canArray implementation code 109

Listing 10: CANremap OBD transmit code 109

Listing 11:CANremap receive Code 110

Listing 1 : EOBD_infotype_interface implementation code

```
001 static uint8 msg_pos,inftyp_len,inftyp_num;
002 uint8  inftyp_idx, inftyp_pos, i;
003 static uint8 cf_wait_count, cf_block_count,cf_index;
004
005
006 // mandatory pattern: (pad with '')
007 //                           "XXXn-YYYYYYYYYYYYYY"
008 static char *EOBD_ECUNAME1 = "ECM\0-AVL RPEMS VCU\0\0";
009 //char EOBD_ECUNAME2[] = "ECM\0-EngineControl\0\0";
010
011 /*
012 Automated Build Time generation:
013 __date__ will expand to a 11-digit string like "Nov 15 2013"
014 __time__ will expand to a 8-digit string like "13:28:23"
015 CALID is 16 characters wide:
016                          "1234567890123456" */
017 static char *EOBD_CALID1 = "RPEMS_08012013_2";
018 static char *EOBD_CALID2 = "Date:" __DATE__; //adjacent strings are automagically
concatenated
019 static char *EOBD_CALID3 = "Time:" __TIME__ "\0\0\0";
020
021 static char *EOBD_CALID[3];
022 EOBD_CALID[0] = EOBD_CALID1;
023 EOBD_CALID[1] = EOBD_CALID2;
024 EOBD_CALID[2] = EOBD_CALID3;
025
026 static char *EOBD_ECUNAME[1];
027 EOBD_ECUNAME[0] = EOBD_ECUNAME1;
028
029 // check if NEW RX Message = 0x02 0x09 0xYY .... (Service 9 request)
030 // This resets the state machine
031 // HINT: CANRX[0] is 2 for physical Addressing!
032
033 if (CANRX[0] &&
034     CANRX[1] == 0x02 &&
035     CANRX[2] == 0x09)
036 {
037     StreamStatus = sm_setup;
038     CANRX[0] = 0; //message = read
039     inftyp = CANRX[3];
040     streamSize = 0;
041 }
042
043 /************************************
044             State Machine
045 ************************************/
```

```
046
047 switch(StreamStatus)
048 {
049 /********************SETUP********************/
050     case sm_setup:
051
052         //determine stream size
053         switch (inftyp)
054         {
055             case 0x00:
056                 // supported INFTYPs
057                 StreamStatus = sm_single;
058                 streamSize = 6; //SID + ITID + 4
059                 break;
060             case 0x04:
061                 my_inftyp = EOBD_CALID;
062                 inftyp_len = 16;
063                 inftyp_num = sizeof(EOBD_CALID)/sizeof(EOBD_CALID[0]);
064                 //SID+ITID+NODI + n*(INFTYP)
065                 streamSize = (inftyp_num * inftyp_len) + 3;
066                 //abort if CALID is not defined
067                 if (streamSize > 3)
068                     StreamStatus = sm_first;
069                 else
070                     StreamStatus = sm_finished;
071                 break;
072             case 0x0A:
073                 my_inftyp = EOBD_ECUNAME;
074                 inftyp_len = 20;
075                 inftyp_num = sizeof(EOBD_ECUNAME)/sizeof(EOBD_ECUNAME[0]);
076                 //SID+ITID+NODI + n*(INFTYP)
077                 streamSize = (inftyp_num * inftyp_len) + 3;
078                 //abort if CALID is not defined
079                 if (streamSize > 3)
080                     StreamStatus = sm_first;
081                 else
082                     StreamStatus = sm_finished;
083                 break;
084             default:
085                 //no other INFTYPs are supported
086                 StreamStatus = sm_finished;
087         }
088
089     break;
090
091 /********************Single Frame********************/
092     case sm_single:
```

```
093         //this only supports INFTYPs up to 0x1F
094         //{TXen}[PCI][0x49][#ITID = 0x00][4-Byte INFTYP][0x55]
095         CANTX[1] = ISO15765_SF | (streamSize & 0xFF);
096         CANTX[2] = 0x49; //SID 9 Response
097         CANTX[3] = 0x00; //INFTYP 0x00
098         CANTX[4] = 0x10; //support 0x04 (CALID)
099         CANTX[5] = 0x40; //support 0x0A (ECUNAME)
100         CANTX[6] = 0x00; //no support 0x11 - 0x18
101         CANTX[7] = 0x00; //no support 0x19 - 0x20
102         CANTX[8] = 0x55; //padding
103         CANTX[0] = 1;   //transmit
104
105         StreamStatus = sm_finished;
106         break;
107
108 /*********************First Frame********************/
109     case sm_first:
110         //{TXen}[PCI][PCI][0x49][#ITID][4 bytes INFTYP]
111         CANTX[1] = ISO15765_FF | ((streamSize >>8) & 0x0F); //highest nibble
112         CANTX[2] = streamSize & 0xFF; //lower two nibbles
113         CANTX[3] = 0x49; //service 9 Response
114         CANTX[4] = inftyp;
115         CANTX[5] = inftyp_num;
116         CANTX[6] = my_inftyp[0][0];
117         CANTX[7] = my_inftyp[0][1];
118         CANTX[8] = my_inftyp[0][2];
119
120         CANTX[0] = 1;
121
122         msg_pos = 6;
123         StreamWaitCounter = 0; //reset timeout
124         StreamStatus = sm_wait_FC; //now wait for the flow control frame
125         break;
126
127 /*********************Flow Control********************/
128     case sm_wait_FC:
129
130         //wait max 2s (200 * 10ms) for F/C frame
131         if (StreamWaitCounter++ == 200)
132         {
133             StreamStatus = sm_finished;
134             break;
135         }
136
137         if (CANRX[0] == 2&&
138             (CANRX[1] & ISO15765_FC) == ISO15765_FC) //is this a new FC frame?
139         {
```

```
140             CANRX[0] = 0;    //ok, we got this
141
142             //===Status===
143             // 0 = OK, go ahead
144             // 1 = WAIT; Tester not ready, wait for another F/C frame
145             // 2 = OVERFLOW; Tester buffer too small. Nothing we can do about that.
146             // N_USDATA.confirm(OVERFLW) is carried out by returning true (=finished)
147             // 3+ = Reserved. Not supported.
148
149             if ((CANRX[1] & 0x0F) == 1)
150             {
151                 break;
152             }
153
154             if ((CANRX[1] & 0x0F) >= 2)
155             {
156                 StreamStatus = sm_finished;
157                 break;
158             }
159
160             //===Block Size:====
161             //0x00: no further F/C just send the rest
162             //0x01-0xFF: nuber of frames before the next F/C negotiation
163
164             Stream_BS = CANRX[2];
165
166             //===Separation Time:===
167             //0x00-0x7F (127d) 0..127ms
168             //0x80-0xF0 -reserved-
169             //0xF1-0xF9 100..900s
170             //0xFA-0xFF -reserved-
171             //since this code does run every 10ms, we can only do multiples of 10ms
ST.
172             //<=11ms: no wait; 11-20ms= wait 1 cycle etc...
173
174             Stream_ST = CANRX[3];
175             if (Stream_ST < 127)
176                 Stream_ST = Stream_ST/10;
177             else if ((Stream_ST > 0xF0) && (Stream_ST < 0xFA))
178                 Stream_ST = 0;
179             else
180                 Stream_ST = 12;
181
182             cf_wait_count = 0;
183             cf_block_count = 0;
184             if (msg_pos == 6)
185                 cf_index = 1; //after the First Frame
```

```
186             else
187                 cf_index = 0;
188
189             StreamStatus = sm_CF;
190         }
191         break;
192
193 /********************Consecutive Frame********************/
194     case sm_CF:
195         //wait lockout
196         if (cf_wait_count++ == Stream_ST)
197             cf_wait_count = 0; //reset wait cycle counter
198         else
199             break;  //wait for next exec cycle
200
201
202         for (i=0; i<7; i++)
203         {
204         inftyp_idx = (msg_pos - 3 )/inftyp_len;
205         inftyp_pos = (msg_pos - 3 )%inftyp_len;
206         CANTX[i+2] = my_inftyp[inftyp_idx][inftyp_pos];
207         msg_pos++;
208         if (msg_pos == streamSize)
209             break;
210         }
211
212         CANTX[1] = ISO15765_CF | cf_index; //PCI
213         CANTX[0] = 1; //transmit
214
215         cf_index = ++cf_index%16; //0,1..15,0,1,...
216         cf_block_count++;
217
218         if (msg_pos == streamSize)
219         {
220             //transfer complete
221             StreamStatus = sm_finished;
222             break;
223         }
224
225         if (Stream_BS && (cf_block_count == Stream_BS))
226         {
227             //reached the Block Size Limit - wait for next F/C Frame
228             StreamWaitCounter = 0; //reset timeout
229             StreamStatus = sm_wait_FC;
230         }
231         break;
232
```

```
233     case sm_finished:
234         return true;
235
236     default:
237         StreamStatus = sm_finished;
238 }
239
240 return false; //transmission not yet finished
```

Listing 2: EOBD_infotype_interface header code

```
1 char **my_inftyp;
2
3 //-----ISO15765-----
4 #define ISO15765_SF 0x00
5 #define ISO15765_FF 0x10
6 #define ISO15765_CF 0x20
7 #define ISO15765_FC 0x30
```

Listing 3: EOBD_remap::update_supported_PIDs_Mode1() implementation code

```
01 uint32 capa_bitfield;
02 uint8 i,j, done, PID_TOP;
03
04 EOBD_max_PID = sizeof(EOBDPIDsAvailable)/sizeof(EOBDPIDsAvailable[0]);
05
06 #define PID_RANGE 0x20
07
08 PID_TOP = (EOBD_max_PID/PID_RANGE) * 0x20; //must be multiple of 0x20
09
10 i = PID_TOP;
11 done = 0;
12
13
14 while (!done)
15 {
16     capa_bitfield = 0x00000000;
17
18     for (j=1; j<=PID_RANGE; j++)
19     {
20         //do not check out of bounds of PIDsAvailable array
21         if (EOBDPIDsAvailable[i + j] &&  ((i+j) < (EOBD_max_PID +1)))
22         {
23             capa_bitfield |= (1 << (PID_RANGE-j));
24         }
25     }
26     if (capa_bitfield)
```

```
27        {
28            EOBDPIDData[i] = capa_bitfield;
29            EOBDPIDsAvailable[i] = 1;
30        }
31
32        if (i == 0)
33        {
34            done = 1;
35        }
36        else
37        {
38            i-=PID_RANGE;
39        }
40 }
```

Listing 4: EOBDpopulateArray::data_update_100ms() header code

```
#define EOBD_VAL(PID, VAL) EOBDPIDData[PID] = VAL << ((4-EOBDPIDSize[PID]) * 8);
```

Listing 5: EOBDpopulateArray::data_update_100ms implementation Code

```
001 /***************************************************************************
002 * This puts the EOBD Mode 1 PID Values ("Live data") into a PID-indexed   *
003 * array. The values are assigned in EOBDremap.                            *
004 *                                                                         *
005 * To add/remove supported PIDs, edit the Data of EOBDPIDsAvailable        *
006 ***************************************************************************/
007
008 // EOBD_VAL(0x00, 0);
009 EOBD_VAL(0x01, EOBD_01_DTC_CNT);
010 // EOBD_VAL(0x02, 0);
011 // EOBD_VAL(0x03, 0);
012 EOBD_VAL(0x04, EOBD_04_LOAD_PCT);
013 EOBD_VAL(0x05, EOBD_05_ECT);
014 // EOBD_VAL(0x06, 0);
[…]
018 // EOBD_VAL(0x0A, 0);
019 EOBD_VAL(0x0B, EOBD_0B_MAP);
020 EOBD_VAL(0x0C, EOBD_0C_RPM);
021 EOBD_VAL(0x0D, EOBD_0D_VSS);
022 // EOBD_VAL(0x0E, 0);
023 // EOBD_VAL(0x0F, 0);
024 // EOBD_VAL(0x10, 0);
025 EOBD_VAL(0x11, EOBD_11_TP);
026 // EOBD_VAL(0x12, 0);
[…]
```

```
035 // EOBD_VAL(0x1B, 0);
036 EOBD_VAL(0x1C, 0x07); //OBD support: EOBD, OBD II
037 // EOBD_VAL(0x1D, 0);
[…]
042 // EOBD_VAL(0x22, 0);
043 EOBD_VAL(0x23, EOBD_23_FRP);
044 // EOBD_VAL(0x24, 0);
[…]
058 // EOBD_VAL(0x32, 0);
059 EOBD_VAL(0x33, EOBD_33_BARO);
060 // EOBD_VAL(0x34, 0);
[…]
073 // EOBD_VAL(0x41, 0);
074 EOBD_VAL(0x42, EOBD_42_VPWR);
075 // EOBD_VAL(0x43, 0);
[…]
078 // EOBD_VAL(0x46, 0);
079 EOBD_VAL(0x47, EOBD_47_TP_B);
080 // EOBD_VAL(0x48, 0);
[…]
155 // EOBD_VAL(0x93, 0);
```

Listing 6: EOBD_DTC_Interface header code

```
01 //--- EEP Memory Map --> see init()---
02 #define EOBD_EEP_SIZE 9
03 #define EOBD_EEP_DTC_SIZE 4
04 #define EOBD_EEP_FF_SIZE 5
05
06 uint8* EEP_EOBD_mem[EOBD_EEP_SIZE];
07
08 #define getbyte(A, B) (((A) >> ((B)*8)) & 0xFF)
09
10 //---All Pages -----
11 #define EEP_CMD 1
12 #define EEP_STATUS 0
13 #define EEP_PAGESIZE 14
14
15 //----Page0-----
16 #define EEP_EOBD_NUMDTC    2
17 #define EEP_EOBD_NUMFF     3
18 #define EEP_EOBD_FIRSTDTC  4
19 #define EEP_EOBD_MIL        5
20 #define EEP_EOBD_FIRSTFF   6
21
22 //-----DTCs----
```

```
23 #define DTC_STARTPAGE 1
24 #define MAX_DTC ((EOBD_EEP_DTC_SIZE * EEP_PAGESIZE) / 2)
25
26 //----FFs----
27 #define FF_STARTPAGE 5
28
29 //-----EEP-----
30 #define EEP_WRITE 8
31 #define EEP_READ  1
32 #define EEP_READOK  1
33 #define EEP_WRITEOK 8
34 #define EEP_READERR  4
35 #define EEP_WRITEERR 128
36
37 //-----ISO15765-----
38 #define ISO15765_SF 0x00
39 #define ISO15765_FF 0x10
40 #define ISO15765_CF 0x20
41 #define ISO15765_FC 0x30
```

Listing 7: EOBD_DTC_Interface implementation code

```
001 /*******************************************************************************
002 *   This is pseudo-file compiled from all the c-functions in EOBD_DTC_Interface        *
003 *   Since ASCET only allows editing of the function BODY, the function wrappers        *
004 *   have been added by hand to show the name and arguments of each function,           *
005 *   replacing ASCET-specific types by more general ones such as "uint" and "bool"      *
006 *   the modifier "private" indicates that the function is declared private in ASCET    *
007 *   and can only be called from inside EOBD_DTC_Interface as "self::some_function()"   *
008 *******************************************************************************/
009
010 void StoreDTC(EOBD_DTC_SystemID System, uint Code, bool enables_MIL, bool store_ff)
011 {
012     int _tmpDTC;
013     int _commit = 0;
014
015     if (EOBD_memRdy)
016     {
017         _tmpDTC = (System << 14) | (Code & 0x3FFF);
018
019         if (enables_MIL)
020         {
021             EOBD_MILStatus = true;
022             _commit = 1;
023         }
024
025         if (store_FF)
```

```
026         {
027             self.StoreFF(_tmpDTC);
028             _commit = 1;
029         }
030
031         if(!self.checkForDuplicateDTC(_tmpDTC))
032         {
033             _nextDTCIdx = self.insertNewDTC();
034
035             _memPage = self.getDTCpage(_nextDTCIdx);
036             _memOffset = self.getDTCoffset(_nextDTCIdx);
037
038             EEP_EOBD_mem[_memPage][_memOffset] = (_tmpDTC >> 8) & 0xFF;
039             EEP_EOBD_mem[_memPage][_memOffset+1] = _tmpDTC & 0xFF;
040
041             _commit = 1;
042         }
043
044         if (_commit)
045             self.commitToEEP();
046     }
047 }
048
049 /***************************************************************************/
050
051 void StoreFF(uint DTC)
052 {
053     uint8 i,j, current_PID,mem_offset,_p,_o,_f;
054
055     if (_EOBD_memRdy)
056     {
057         //get next "free" frame / memory location
058         _f = self.insertNewFF();
059         mem_offset = self.getFFTotalOffset(_f,0);
060
061         //write the responsible DTC
062         _p = self.getFFpage(mem_offset);
063         _o = self.getFFoffset(mem_offset++);
064         EEP_EOBD_mem[_p][_o] = (DTC >> 8) & 0xFF;
065
066         _p = self.getFFpage(mem_offset);
067         _o = self.getFFoffset(mem_offset++);
068         EEP_EOBD_mem[_p][_o] = DTC & 0xFF;
069
070         //Dump the PIDs in the FF Memory Area
071         for (i=0; EOBDFFPIDs[i]; i++)
072         {
```

```
073             current_PID = EOBDFFPIDs[i];
074
075             //for each PID, write the bytes
076             for (j=0;j<EOBDPIDSize[current_PID];j++)
077             {
078                 _p = self.getFFpage(mem_offset);
079                 _o = self.getFFoffset(mem_offset++);
080         //the PID Data is left-aligned in a 32 bit field -> MSB = (uint32)>>24 !
081                 EEP_EOBD_mem[_p][_o] = getbyte(EOBDPIDData[current_PID],(3-j));
082             }
083         }
084
085         //self.commitToEEP();
086     }
087 }
088
089 /***************************************************************************/
090
091 void clearDTCs()
092 {
093     //since DTCs cannot be deletet individually, it is OK to just zero their count.
094     //No management of the actual DTC storage has to be done.
095
096     int i,j;
097
098     EOBD_numDTC = 0;
099     _EOBD_firstDTCIdx = 0;
100     EOBD_MILStatus = false;
101     EOBD_numFF = 0;
102     _EOBD_firstFFIdx = 0;
103
104     //If the Paramater is selected, the memory Area is "formatted".
105     //if(EOBD_DTCFullerease == true)
106     {
107         for (i = 0; i < EOBD_EEP_SIZE; i++)
108         {
109             for (j = 2; j<16; j++)
110             {
111                 EEP_EOBD_mem[i][j] = 0;
112             }
113         }
114         passed_FE = true;
115     }
116
117     //immediately commit, so errors don't "come back"
118     self.commitToEEP();
119 }
```

```
120
121 /*****************************************************************************/
122
123 void commitToEEP()
124 {
125     //commits Data from RAM to EEP
126
127     uint8 i;
128
129     EEP_EOBD_mem[0][EEP_EOBD_NUMDTC] = EOBD_numDTC;
130     EEP_EOBD_mem[0][EEP_EOBD_FIRSTDTC] = _EOBD_firstDTCIdx;
131     EEP_EOBD_mem[0][EEP_EOBD_MIL] = EOBD_MILStatus;
132     EEP_EOBD_mem[0][EEP_EOBD_NUMFF] = _EOBD_numFF;
133     EEP_EOBD_mem[0][EEP_EOBD_FIRSTFF] = _EOBD_firstFFIdx;
134
135     for (i=0; i<EOBD_EEP_SIZE; i++)
136     {
137         EEP_EOBD_mem[i][EEP_CMD] = EEP_WRITE;
138     }
139 }
140
141 /*****************************************************************************/
142
143 void EOBD_DTC_MGMT_100ms()
144 {
145     // copy EEP stuff to RAM as soon as the EEP becomes available
146     // and skip the block after that (!_EOBD:memRdy)
147
148     if((!EOBD_memRdy) && self.DTCMemCheck())
149     {
150         EOBD_numDTC =       EEP_EOBD_mem[0][EEP_EOBD_NUMDTC];
151         _EOBD_firstDTCIdx = EEP_EOBD_mem[0][EEP_EOBD_FIRSTDTC];
152         EOBD_MILStatus =    EEP_EOBD_mem[0][EEP_EOBD_MIL];
153         EOBD_numFF =        EEP_EOBD_mem[0][EEP_EOBD_NUMFF];
154         _EOBD_firstFFIdx =  EEP_EOBD_mem[0][EEP_EOBD_FIRSTFF];
155
156         EOBD_memRdy = true;
157     }
158
159     DTCrunning += 1;
160
161     sizeof_0 = self.getFFSpecialPID(0x00);
162     sizeof_1 = self.getFFSpecialPID(0x20);
163     sizeof_2 = self.getFFSpecialPID(0x40);
164     sizeof_3 = self.getFFSpecialPID(0x80);
165 }
166
```

```
167 /************************************************************************/
168
169 void init()
170 {
171     //This maps "real" EEP Pages in the EOBD Memory Map
172     //The EEP Pages don't need to be consecutive.
173
174     uint8 i;
175
176     EEP_EOBD_mem[0] = eepPage_2;    //bookkeeping
177     EEP_EOBD_mem[1] = eepPage_3;    // DTCs 0-6
178     EEP_EOBD_mem[2] = eepPage_4;    // DTCs 7-13
179     EEP_EOBD_mem[3] = eepPage_5;    // DTCs 14-20
180     EEP_EOBD_mem[4] = eepPage_6;    // DTCs 21-27
181     EEP_EOBD_mem[5] = eepPage_7;    // FF Data
182     EEP_EOBD_mem[6] = eepPage_8;    // FF Data
183     EEP_EOBD_mem[7] = eepPage_9;    // FF Data
184     EEP_EOBD_mem[8] = eepPage_10;   // FF Data
185
186
187    //calculate the Size of a EOBD FF
188    //it is the sum of the PID sizes contained in it + 2 Bytes for the associated DTC
189
190     _EOBD_FFSize = 2;
191
192     for (i=0; EOBDFFPIDs[i]; i++)
193     {
194         _EOBD_FFSize += EOBDPIDSize[EOBDFFPIDs[i]];
195     }
196
197     //calculate the maximum amount of FFs that can be stored in the allocated memory
198     //this might be 0 if the list of PIDs is too long
199
200     _EOBD_MaxFF = (EOBD_EEP_FF_SIZE*EEP_PAGESIZE)/_EOBD_FFSize;
201 }
202
203 /************************************************************************/
204
205 bool DTCMessageStream(uint8[] CANRXData, uint8[]CANTXData)
206 {
207     uint16 tmp_dtc;
208     uint8 i;
209
210     static uint8 cf_wait_count, cf_block_count, half_dtc_pending, half_dtc_byte,
cf_index;
211
212     // check if NEW RX Message = 0x01 0x02 .... (Service 2 request)
```

```
213     // This resets the state machine
214     // HINT: CANRXData[0] is 2 for physical Adressing!
215
216     if (CANRXData[0] &&
217         CANRXData[1] == 0x01 &&
218         CANRXData[2] == 0x03)
219     {
220         StreamStatus = sm_setup;
221         CANRXData[0] = 0; //message = read
222     }
223
224     /*************************************
225                 State Machine
226     *************************************/
227
228     switch(StreamStatus)
229     {
230     /*********************SETUP*********************/
231         case sm_setup:
232             //we can transmit up to 2 complete DTCs in a SF, for
233             //more we have to do a segmented transfer
234             if (_EOBD_numDTC <= 2)
235             {
236                 StreamStatus = sm_single;
237                 DTCStreamNumDTC = _EOBD_numDTC;
238             }
239             else
240                 StreamStatus = sm_first;
241             break;
242
243     /*********************Single Frame*********************/
244         case sm_single:
245             //{TXen}[PCI][0x43][#DTC][DTC1L][DTC1H][DTC2l][DTC2H][0x55]
246             CANTXData[1] = ISO15765_SF | (2 + DTCStreamNumDTC * 2);
247             CANTXData[2] = 0x43; //SID 3 Response
248             CANTXData[3] = DTCStreamNumDTC;
249             for (i=0;i < DTCStreamNumDTC; i++)
250             {
251                 tmp_dtc = self.ReadDTCFromEEP(i);
252                 CANTXData[4 + 2*i] = (tmp_dtc >> 8) & 0xFF;
253                 CANTXData[5 + 2*i] = tmp_dtc & 0xFF;
254             }
255             CANTXData[8] = 0x55;
256             CANTXData[0] = 1;
257
258             StreamStatus = sm_finished;
259             break;
```

```
260
261     /********************First Frame********************/
262         case sm_first:
263             //{TXen}[PCI][PCI][0x43][#DTC][DTC1L][DTC1H][DTC21][DTC2H]
264             DTCStreamNumDTC = _EOBD_numDTC;
265             DTCStreamLen = 2 + DTCStreamNumDTC * 2; //1DTC = 2 bytes + [0x43][#DTC]
266             CANTXData[1] = ISO15765_FF | ((DTCStreamLen >>8) & 0x0F); //highest
nibble
267             CANTXData[2] = DTCStreamLen & 0xFF; //lower two nibbles
268             CANTXData[3] = 0x43; //service 3 Response
269             CANTXData[4] = DTCStreamNumDTC;
270
271             DTCStreamNextDTC = 0;
272             for (i=0;i < 2; i++)
273             {
274                 tmp_dtc = self.ReadDTCFromEEP(DTCStreamNextDTC++);
275                 CANTXData[5 + 2*i] = (tmp_dtc >> 8) & 0xFF; //MSB first
276                 CANTXData[6 + 2*i] = tmp_dtc & 0xFF;
277             }
278             CANTXData[0] = 1; //transmit
279
280             half_dtc_pending = 0; // we have to do that here, since F/C can also ba
after a "half" DTC
281             cf_index = 1; //this is the 0th frame...
282
283             StreamWaitCounter = 0; //reset timeout
284             StreamStatus = sm_wait_FC; //now wait for the flow control frame
285             break;
286
287     /********************Flow Control********************/
288         case sm_wait_FC:
289
290             //wait max 2s (200 * 10ms) for F/C frame
291             if (StreamWaitCounter++ == 200)
292             {
293                 StreamStatus = sm_finished;
294                 break;
295             }
296
297             if (CANRXData[0] == 2&&
298                 (CANRXData[1] & ISO15765_FC) == ISO15765_FC) //is this a new FC
frame?
299             {
300                 CANRXData[0] = 0;   //ok, we got this
301
302                 //===Status===
303                 // 0 = OK, go ahead
304                 // 1 = WAIT; Tester not ready, wait for another F/C frame
```

```
305                 // 2 = OVERFLOW; Tester buffer too small. Nothing we can do about
that.
306                 // N_USDATA.confirm(OVERFLW) is carried out by returning true
(=finished)
307                 // 3+ = Reserved. Not supported.
308
309                 if ((CANRXData[1] & 0x0F) == 1)
310                 {
311                     break;
312                 }
313
314                 if ((CANRXData[1] & 0x0F) >= 2)
315                 {
316                     StreamStatus = sm_finished;
317                     break;
318                 }
319
320
321                 //===Block Size:====
322                 //0x00: no further F/C just send the rest
323                 //0x01-0xFF: nuber of frames before the next F/C negotiation
324
325                 Stream_BS = CANRXData[2];
326
327                 //===Separation Time:===
328                 //0x00-0x7F (127d) 0..127ms
329                 //0x80-0xF0 -reserved-
330                 //0xF1-0xF9 100..900s
331                 //0xFA-0xFF -reserved-
332                 //since this code does run every 10ms, we can only do multiples of 10ms ST.
333                 //<=11ms: no wait; 11-20ms= wait 1 cycle etc...
334
335                 Stream_ST = CANRXData[3];
336                 if (Stream_ST < 127)
337                     Stream_ST = Stream_ST/10;
338                 else if ((Stream_ST > 0xF0) && (Stream_ST < 0xFA))
339                     Stream_ST = 0;
340                 else
341                     Stream_ST = 12;
342
343                 cf_wait_count = 0;
344                 cf_block_count = 0;
345                 if (DTCStreamNextDTC == 2)
346                     cf_index = 1; //after the First Frame
347                 else
348                     cf_index = 0;
349
350                 StreamStatus = sm_CF;
```

```
351             }
352             break;
353
354     /********************Consecutive Frame********************/
355         case sm_CF:
356             //wait lockout
357             if (cf_wait_count++ == Stream_ST)
358                 cf_wait_count = 0; //reset wait cycle counter
359             else
360                 break;  //wait for next exec cycle
361
362             //try to write up to 3 DTC in Message - if there is a half DTC pending,
363             //offset them by 1 byte to leave room for that.
364             //if the end of the DTCs is reached before, break out of the loop
365             for (i=0;i < 3; i++)
366             {
367                 if (DTCStreamNumDTC == DTCStreamNextDTC)
368                     break; //no more DTCs to send, maybe a half one
369
370                 tmp_dtc = self.ReadDTCFromEEP(DTCStreamNextDTC++);
371                 CANTXData[2 + 2*i + half_dtc_pending] = (tmp_dtc >> 8) & 0xFF; //MSB
First
372                 CANTXData[3 + 2*i + half_dtc_pending] = tmp_dtc & 0xFF;
373
374             }
375
376             //is there still "half" a DTC to transmit?
377             if (half_dtc_pending)
378             {
379                 CANTXData[2] = half_dtc_byte;
380                 half_dtc_pending = 0;
381             }
382             //we didnt have a half left so we produce one if needed
383             else if (!(DTCStreamNumDTC == DTCStreamNextDTC))
384             {
385                 tmp_dtc = self.ReadDTCFromEEP(DTCStreamNextDTC++);
386                 CANTXData[8] = (tmp_dtc >> 8) & 0xFF; //MSB first
387                 half_dtc_byte = tmp_dtc & 0xFF;
388                 half_dtc_pending = 1;
389             }
390
391             CANTXData[1] = ISO15765_CF | cf_index; //PCI
392             CANTXData[0] = 1; //transmit
393
394             cf_index = ++cf_index%16; //0,1..15,0,1,...
395             cf_block_count++;
396
```

```
397             if ((DTCStreamNumDTC == DTCStreamNextDTC) & !half_dtc_pending)
398             {
399                 //transfer complete
400                 StreamStatus = sm_finished;
401                 break;
402             }
403 
404             if (Stream_BS && (cf_block_count == Stream_BS))
405             {
406                 //reached the Block Size Limit - wait for next F/C Frame
407                 StreamWaitCounter = 0; //reset timeout
408                 StreamStatus = sm_wait_FC;
409             }
410 
411             break;
412 
413         case sm_finished:
414             return true;
415 
416         default:
417             StreamStatus = sm_finished;
418     }
419 
420     return false; //transmission not yet finished
421 }
422 
423 /***************************************************************************/
424 
425 getFFPID(uint frame, uint PID, uint8[] CANTXdata)
426 {
427     //-->See headers!!
428 
429     uint32 _tmpPID =0;
430     uint8 i,current_PID,mem_offset,_p,_o;
431 
432     if (_EOBD_memRdy)
433     {
434         mem_offset = 0;
435         _tmpPID = 0;
436 
437         switch (PID)
438         {
439             case 0x00:
440             case 0x20:
441             case 0x40:
442             case 0x60:
443             case 0x80:
```

```
444                 _tmpPID = self.getFFSpecialPID(PID);
445                 if (_tmpPID)
446                 {
447                     //only set current_PID if the returned value (supported PIDs) is
non-zero
448                     //empty support bitfields will be reported as not supported and
therefore wil not be responded to
449                     current_PID = PID;
450                 }
451                 break;
452
453             default:
454                 if (PID == 0x02)
455                 {
456                     //PID 0x02 is the FF DTC, which is at offset 0
457                     current_PID = 0x02;
458                 }
459                 else
460                 {
461                     //the first 2 bytes are for the FF DTC
462                     mem_offset = 2;
463                     //calculate the PID memory position inside a FF
464                     //this assumes that the PID exists -  The tester is required by
ISO15031
465                     //to query this information before making a request.
466
467                     current_PID = 0;
468                     for (i=0; EOBDFFPIDs[i]; i++) //traverse until we hit a 0
469                     {
470                         current_PID = EOBDFFPIDs[i];
471                         if (current_PID == PID)
472                             break;
473                         else
474                         {
475                             mem_offset += EOBDPIDSize[current_PID];
476
477                             //if the next PID in the list is 0 and the current is
not the one we
478                             //want, it is not in the list, therefore not supported
479                             if (EOBDFFPIDs[i+1] == 0)
480                             {
481                                 //indicate that the PID is not in the list
482                                 current_PID = 0;
483                             }
484                         }
485                     }
486
487                 }
```

```
488
489                 //only if supported
490                 if(current_PID)
491                 {
492                     //get the "real" position of the data in the FF/DTC Memory blob
493                     mem_offset = self.getFFTotalOffset(frame,mem_offset);
494
495                     //bytewise assemble PID from EEP Memory
496                     for (i=0;i<EOBDPIDSize[PID];i++)
497                     {
498                         _p = self.getFFpage(mem_offset);
499                         _o = self.getFFoffset(mem_offset);
500                         _tmpPID <<= 8;
501                         _tmpPID |= EEP_EOBD_mem[_p][_o] & 0xFF;
502                         mem_offset++;
503                     }
504                 }
505         }
506
507         if (current_PID || (PID == 0))
508         {
509             //left-align PID data int the CAN Msg
510
511             CANTXdata[1] = ISO15765_SF | (EOBDPIDSize[PID] + 3);
512             CANTXdata[2] = 0x42; //0x02 + 0x40
513             CANTXdata[3] = PID;
514             CANTXdata[4] = frame;
515             CANTXdata[5] = 0x55;
516             CANTXdata[6] = 0x55;
517             CANTXdata[7] = 0x55;
518             CANTXdata[8] = 0x55;
519
520             for(i=0; i < EOBDPIDSize[PID]; i++)
521             {
522                 CANTXdata[i+5] = (_tmpPID >> ((EOBDPIDSize[PID] - 1-i)*8)) & 0xFF;
523             }
524
525             CANTXdata[0] = 1;
526         }
527         else
528         {
529             CANTXdata[0] = 0;
530         }
531     }
532 }
533
534 /***************************************************************************/
```

```
535
536 uint getDTCpage(uint DTCAdr)
537 {
538     //zero-indexed!
539     //The DTCAdr Argument is the ABSOLUTE Adress of the DTC in the DTC Storage
540     //Memory space. It does not account for wrap-around etc.
541
542     if (DTCAdr <= MAX_DTC)
543         return (((DTCAdr) / (EEP_PAGESIZE/2)) + DTC_STARTPAGE);
544     else
545         return 1;
546 }
547
548 /***************************************************************************/
549
550 uint getDTCoffset(uint DTCAdr)
551 {
552     //zero-indexed!
553     //The DTCAdr Argument is the ABSOLUTE Adress of the DTC in the DTC Storage
554     //Memory space. It does not account for wrap-around etc.
555     //(EEP_PAGESIZE/2) = number of DTCs (2 byte) per Page - works only for even page
sizes!
556     // .. * 2 converts the DTC number inside the page to the byte offset
557     // add 2 since the usable offsets start at 2. (0,1) are command & control.
558
559     if (DTCAdr <= MAX_DTC)
560         return (((DTCAdr) % (EEP_PAGESIZE/2) * 2) + 2);
561     else
562         return 2;
563 }
564
565 /***************************************************************************/
566
567 uint getFFTotalOffset(uint frame, uint mem_offset)
568 {
569     // get the page on which a certain memory location (offset)
570     // within a certain frame# (frame) resides
571
572     uint8 total_offset, real_frame;
573
574
575     //calculate real frame locaion
576     real_frame=((_EOBD_firstFFIdx + frame) % _EOBD_MaxFF);
577
578     total_offset = real_frame * _EOBD_FFSize + mem_offset;
579
580     return (total_offset);
```

```
581 }
582
583 /***************************************************************************/
584
585 bool DTCMemCheck(void)
586 {
587     int i;
588     int _memOK = 1;
589
590     for (i=0; i < EOBD_EEP_SIZE; i++)
591     {
592         if (!(EEP_EOBD_mem[i][EEP_STATUS] & (EEP_READOK|EEP_WRITEOK)))
593             _memOK = 0;
594     }
595
596     if (_memOK)
597         return true;
598     else
599         return false;
600 }
601
602 /***************************************************************************/
603
604 bool checkForDuplicateDTC(uint DTC)
605 {
606     uint16 i, _tmp_adr, _tmp_DTC;
607     int _found = 0;
608
609     for (i=0; i < EOBD_numDTC; i++)
610     {
611         _tmp_adr = self.findDTC(i);
612         _memPage = self.getDTCpage(_tmp_adr);
613         _memOffset = self.getDTCoffset(_tmp_adr);
614         _tmp_DTC = ((EEP_EOBD_mem[_memPage][_memOffset])<<8) |
(EEP_EOBD_mem[_memPage][_memOffset+1]);
615
616         DTCLookupTmpDTC = _tmp_DTC;
617         DTClookupiterations = i+1;
618
619         if (_tmp_DTC == DTC)
620             return true;
621     }
622
623     DTCLookupNotFound += 1;
624
625     return false;
626 }
```

```
627
628 /**************************************************************************/
629
630 uint insertNewDTC(void)
631 {
632     //Rollover logic:
633     //The DTCs are stored in a "ring"
634     //_numDTC is the actual number of saved DTCs
635     //internally, the DTCs are stored on a zero-based index
636     //if _numDTC (the number of stored DTCs) reaches MAX_DTC, the oldest DTCs get
overwritten.
637     //_numDTCs then stays at MAX_DTC and _firstDTCIdx starts moving from 0
638
639     uint8 _next_free;
640
641     if (EOBD_numDTC >= MAX_DTC)
642     {
643         _next_free   = _EOBD_firstDTCIdx++;
644         _EOBD_firstDTCIdx %= MAX_DTC;
645     }
646     else
647     {
648         _next_free = _EOBD_numDTC++;
649     }
650
651     return _next_free;
652 }
653
654 /**************************************************************************/
655
656 uint insertNewFF(void)
657 {
658     //Rollover logic:
659     //The DTCs are stored in a "ring"
660     //_numDTC is the actual number of saved DTCs
661     //internally, the DTCs are stored on a zero-based index
662     //if _numDTC (the number of stored DTCs) reaches MAX_DTC, the oldest DTCs get
overwritten.
663     //_numDTCs then stays at MAX_DTC and _firstDTCIdx starts moving from 0
664
665     uint8 _next_free;
666
667     if (EOBD_numDTC >= MAX_DTC)
668     {
669         _next_free   = _EOBD_firstDTCIdx++;
670         _EOBD_firstDTCIdx %= MAX_DTC;
671     }
672     else
```

```
673     {
674         _next_free = _EOBD_numDTC++;
675     }
676
677     return _next_free;
678 }
679
680 /*************************************************************************/
681
682 uint findDTC(uint DTCnum)
683 {
684     // Rollover logic:
685     // The DTCs are stored in a "ring"
686     // _numDTC is the actual number of saved DTCs
687     // internally, the DTCs are stored on a zero-based index
688     // if _numDTC (the number of stored DTCs) is MAX_DTC, the oldest DTCs got
overwritten.
689     // _numDTCs then stays at MAX_DTC and _firstDTCIdx starts moving from 0 - So the
"real"
690     // DTC Address is offset by _firstDTCIdx and wraps around the top.
691
692     int _real_adr;
693
694     if (_EOBD_numDTC >= MAX_DTC)
695     {
696         _real_adr = (DTCnum + _EOBD_firstDTCIdx) % MAX_DTC;
697     }
698     else
699     {
700         _real_adr = (DTCnum);
701     }
702
703     return _real_adr;
704 }
705
706 /*************************************************************************/
707
708 uint ReadDTCFromEEP(int DTCnum)
709 {
710     uint16 _tmp_adr, _tmp_DTC;
711
712     if(EOBD_memRdy)
713     {
714         if (DTCnum < EOBD_numDTC)
715         {
716             _tmp_adr = self.findDTC(DTCnum);
717             _memPage = self.getDTCpage(_tmp_adr);
718             _memOffset = self.getDTCoffset(_tmp_adr);
```

```
719
720             _tmp_DTC = ((EEP_EOBD_mem[_memPage][_memOffset])<<8) |
(EEP_EOBD_mem[_memPage][_memOffset+1]);
721             return _tmp_DTC;
722         }
723         else
724             return 0;
725     }
726     else
727         return 0;
728 }
729
730 /***************************************************************************/
731
732 uint getFFpage(uint abs_mem_offset)
733 {
734     return ((abs_mem_offset / EEP_PAGESIZE) + FF_STARTPAGE);
735 }
736
737 /***************************************************************************/
738
739 uint getFFoffset(uint abs_mem_offset)
740 {
741     //add 2, since the usable offsets start at 2.
742     //0,1 are status/command bytes
743
744     return ((abs_mem_offset % EEP_PAGESIZE) + 2);
745 }
746
747 /***************************************************************************/
748
749 uint getFFSpecialPID(uint PID)
750 {
751     uint8 i;
752     uint32 tmp_pid = 0;
753
754     switch (PID)
755     {
756         //case 0x00,0x20,0x40,0x60,0x80:
757         //these PIDs bit-encode the other available PIDs
758         //so we walk through the EOBDFFPIDs list and set the bits accordingly
759         case 0x00:
760             //PID 2 is always supported
761             tmp_pid = 0x40000000;
762         case 0x20:
763         case 0x40:
764         case 0x60:
```

```
765         case 0x80:
766             // The last byte of EOBDFFPIDs has to be 0, or this will hang
767             for (i=0; EOBDFFPIDs[i]; i++)
768             {
769                 //if PID is within the PID's encoded range (PID+1...PID+0x20)
770                 //flip on a bit at it's relative position (MSB = bit0)
771                 if ((EOBDFFPIDs[i] > PID) && (EOBDFFPIDs[i] <= (PID + 0x20)))
772                 {
773                     tmp_pid |= 1<<(32 - EOBDFFPIDs[i] + PID);
774                 }
775 
776                 //if at least one PID above the current range exists,
777                 //set the last bit to indicate that another PID index is available.
778                 if(EOBDFFPIDs[i] > (PID + 0x20))
779                     tmp_pid |= 0x00000001;
780             }
781             break;
782         default:
783             break;
784     }
785 
786     return tmp_pid;
787 }
```

Listing 8: EOBD_bytesplit implementation code

```
01 return  (value >> (numbyte*8)) & 0x000000FF;
```

Listing 9:EOBD_assemble_canArray implementation code

```
01 //the CANarrayPtr Argument provides a pointer to the Array's
02 //Memory Location when connected to the GET port of an Array
03 
04 CANarrayPtr[0] = NewData;
05 CANarrayPtr[1] = Data0;
06 CANarrayPtr[2] = Data1;
07 CANarrayPtr[3] = Data2;
08 CANarrayPtr[4] = Data3;
09 CANarrayPtr[5] = Data4;
10 CANarrayPtr[6] = Data5;
11 CANarrayPtr[7] = Data6;
12 CANarrayPtr[8] = Data7;
```

Listing 10: CANremap OBD transmit code

```
01 /***************************************************************************
02 * EOBD TxMOb 0x7E8
```

```
03 ***************************************************************************/
04 if (EOBD_TXCAN[0]) //NewMsg Flag
05 {
06  DataT96[0] = EOBD_TXCAN[1];
07  DataT96[1] = EOBD_TXCAN[2];
08  DataT96[2] = EOBD_TXCAN[3];
09  DataT96[3] = EOBD_TXCAN[4];
10  DataT96[4] = EOBD_TXCAN[5];
11  DataT96[5] = EOBD_TXCAN[6];
12  DataT96[6] = EOBD_TXCAN[7];
13  DataT96[7] = EOBD_TXCAN[8];
14
15  EOBD_TXCAN[0] = 0;
16  EOBD_TXFlag = 1;
17 }
```

Listing 11:CANremap receive Code

```
01 /***************************************************************************
02 * ID 7DFh - EOBD Functional RX Address
03 ***************************************************************************/
04
05 if (EOBD_RXFlag_F)
06 {
07  EOBD_RXCAN[1]     = DataR112[0];
08  EOBD_RXCAN[2]     = DataR112[1];
09  EOBD_RXCAN[3]     = DataR112[2];
10  EOBD_RXCAN[4]     = DataR112[3];
11  EOBD_RXCAN[5]     = DataR112[4];
12  EOBD_RXCAN[6]     = DataR112[5];
13  EOBD_RXCAN[7]     = DataR112[6];
14  EOBD_RXCAN[8]     = DataR112[7];
15  EOBD_RXCAN[0]     = 1;  //NewMsg Flag
16  EOBD_RXFlag_F     = 0;
17 }
18
19 /***************************************************************************
20 * ID 7Enh - EOBD Physical RX Address
21 ***************************************************************************/
22
23 if (EOBD_RXFlag_P)
24 {
25  EOBD_RXCAN[1]     = DataR113[0];
26  EOBD_RXCAN[2]     = DataR113[1];
27  EOBD_RXCAN[3]     = DataR113[2];
28  EOBD_RXCAN[4]     = DataR113[3];
```

```
29  EOBD_RXCAN[5]      = DataR113[4];
30  EOBD_RXCAN[6]      = DataR113[5];
31  EOBD_RXCAN[7]      = DataR113[6];
32  EOBD_RXCAN[8]      = DataR113[7];
33  EOBD_RXCAN[0]      = 2;  //NewMsg Flag
34  EOBD_RXFlag_P      = 0;
35 }
```