TEKNISKA HÖGSKOLAN

HÖGSKOLAN I JÖNKÖPING

Bootloader with reprogramming functionality for electronic control units in vehicles: Analysis, design and Implementation

David Pehrsson

Jesús Garza

EXAM WORK 2012

ELECTRICAL ENGINEERING

TEKNISKA HÖGSKOLAN

HÖGSKOLAN I JÖNKÖPING

Examiner: Shashi Kumar

Supervisor: Alf Johansson

Scope: 30 credits (second cycle)

Date: 2012-12-18

Acknowledgements

We would like to thank everyone at QRTECH for the help they have given us and we would especially like to thank Andreas Käck for the technical support he provided when we needed it. We will also like to thank Lars Carlsson on QRTECH for all the help regarding the master thesis. We would also like to thank the teachers at JTH, Alf Johansson and Shashi Kumar for their support.

Abstract

In an automotive context today's need of testing functions while in factory, correcting faults in the workshop or adding extra value in the aftermarket makes it very important to easily be able to download new software to the electronic control units in vehicles. In the platform for standard automotive software development called AUTOSAR, two known protocols are presented to specify the procedure on how to implement this download operation: Unified Diagnostic Services (UDS) and the Universal Measurement and Calibration Protocol (XCP).

However the part of the UDS and XCP standards that is about reprogramming is not completely a part of the AUTOSAR standard yet. In this thesis, UDS and XCP have been compared to evaluate which of the two that has most support in AUTOSAR today and are most likely to be fully integrated into AUTOSAR in the future. Since UDS already has support in AUTOSAR for some of the functions needed for reprogramming and because of the fact that UDS is a part of the extensively used On-board Diagnostic standard (OBD-II), UDS is chosen to be the most suitable protocol for implementing reprogramming functionality according to AUTOSAR.

A bootloader with the ability to download data has been developed using only relevant functions from UDS and following the AUTOSAR specifications where it is applicable.

Sammanfattning

För att kunna testa fordonsfunktioner i fabriken, åtgärda mjukvarufel under service eller för att uppgradera fordonet med nya funktioner är det viktigt att kunna ladda ner ny mjukvara till fordonets styrsystem. Den standardiserade mjukvaruplattformen för fordonsindustrin, AUTOSAR, innehåller två protokoll som båda specificerar hur mjukvara kan laddas ner: Unified Diagnostic Services (UDS) och Universal Measurement and Calibration Protocol (XCP).

Tyvärr är de delarna av UDS och XCP som beskriver mjukvarunerladdning inte en del av AUTOSAR än. I det här examensarbetet har UDS och XCP jämförts för att utvärdera vilken av de båda som i dagsläget har störst stöd för nerladdning av mjukvara i AUTOSAR och vilken som troligast kommer att bli en del av AUTOSAR i framtiden. Eftersom AUTOSAR redan stödjer några av de funktioner i UDS som behövs för nerladdning av mjukvara samt på grund av att UDS är en del av branschstandarden för fordonsdiagnostik OBD-II, har UDS valts som den mest lämpade att i dagsläget användas för att implementera nerladdning av mjukvara enligt AUTOSAR.

En bootloader som stödjer nerladdning av mjukvara via UDS har sedan implementerats enligt AUTOSAR-specifikationen så långt som möjligt.

Keywords

AUTOSAR

UDS

XCP

Bootloader

Reprogramming

ECU

Software Downloads

CAN

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AUTOSAR	Automotive Open System Architecture
BAM	Boot Assist Module
CAN	Controller Area Network
ExtRAM	External Random Access Memory
MCU	Microcontroller Unit
MMU	Memory Management Unit
PBL	Primary Bootloader
PCI	Protocol Control Information
SID	Service Identifier
PDU	Protocol Data Unit
CANTp	CAN Transport Protocol
RCHW	Reset Configuration Half Word
ROM	Read-Only Memory
SBL	Secondary Bootloader
SPE	Signal Processing Extension
SRAM	Internal Static Random Access Memory
UDS	Unified Diagnostic Services
XCP	Universal Measurement and Calibration Protocol
DCM	Diagnostic Communication Manager
PPC	PowerPC
EABI	Embedded Application Binary Interface
Tx	Process of data transmission
Rx	Process of data reception

1 Introduction

1.1 Background

Within the automotive industry there is a recurrent need of loading new application software to the embedded units responsible of vehicle functions, whether it is under development, during maintenance or as a post-sales upgrade. For this to be possible it is required that these embedded units (also known as "ECU") support reprogramming through a standard communication interface such as CAN, FlexRay, LIN, Ethernet, etc.

In recent years, several car manufacturers, suppliers and tool developers have adopted a standard for the management of vehicle functions within both future applications and standard software modules; such standard is named AUTOSAR (Automotive Open System Architecture). One case of software module management is precisely the area of ECU reprogramming. The AUTOSAR standard, however, does not exactly specify how to download a new application program to a target processor within a vehicle's Electronic Control Unit (ECU).

AUTOSAR supports two standards for diagnostic and calibration of vehicle parameters (UDS and XCP, respectively) which can also provide resources for reprogramming an ECU's processor. An analysis of compatibility between each of those standards and AUTOSAR will be done for selecting the best of both. Such selection will lead to the development of a bootloader that will support system start-up and software functionality for reprogramming the QR5567 platform via the standard communication interface CAN. This development shall meet the requirements imposed by AUTOSAR where it is applicable.

The functionality of the bootloaders will provide a solution that can be applied to load and start software onto the platform QR5567, which can then be used for further prototyping. This development will intend to satisfy the need expressed above since its operation can emulate the process of reprogramming a car's ECU while still during development in the factory, or while correcting a failure in a service workshop or even when some extra functionality is provided in the aftermarket.

1.2 Purpose and research questions

The purpose is to analyse the communication protocols XCP and UDS from an AUTOSAR perspective to determine which one is most suitable to use for reprogramming of an ECU that are developed according to AUTOSAR. After the analysis is done a bootloader that supports the more appropriate standard is to be implemented on the development platform QR5567 according to the AUTOSAR specifications as far as possible.

1.3 Delimitations

In the analysis of communication protocols UDS and XCP only functions regarding reprogramming will be covered, other part of UDS and XCP will not be taken into consideration when comparing them. Only relevant parts regarding reprogramming of AUTOSAR will be implemented. This report will only focus on CAN as the communication interface, even though other communication interfaces are supported in hardware, and XCP, UDS and AUTOSAR.

1.4 Outline

The first part of the report, theoretical background, covers the bootloader concept and the concepts and the standards CAN, AUTOSAR, UDS and XCP. The next part, Method and implementation, describes the features of UDS from an AUTOSAR perspective and how UDS and AUTOSAR are used in the implementation. In the third part, Findings and Analysis, the outcome of the functions of the software that has been implemented is presented. The last part, Discussion and Conclusions, includes a comment on the coverage of bootloaders provided by AUTOSAR, summarizes the work and proposes how it can be further developed.

2 Theoretical background

2.1 Bootloader

Among the multiple conceptions that exist regarding the definition of a bootloader one common approach is that it is considered to be a fixed piece of software or firmware residing at least partially in the non-volatile memory area of a microprocessor, such as ROM or Flash. The "firm" condition of the bootloader is based on the idea that once designed and developed it's not supposed to be object of many changes or maintenance during the processor lifetime as an application program would eventually undergo.

The different views that exist towards the features and operation that a bootloader should perform are often motivated, for instance, by the following factors:

- The resources eventually needed by a running application (time, memory, interrupts)
- The resources supported by each specific MCU (types of memories, access to special function registers, interrupt stack).
- The amount of memory available for storing initialization code.

Despite these variations it's a common practice to implement the bootloader in such a fashion that it starts running right after power-up, whether coming from a system software reset, external induced reset (incoming signal or command via communication interface or event-sensed configured), manual reset or by just applying power supply to the processor to turn it on.

Usually embedded processors fetch and execute code from the reset vector at a defined address in ROM or Flash, to further jump to another section of memory where the initialization code resides. This is done to keep the reset vector small.

Whether it is due the requirements imposed by an application or the capabilities of a specific processor, at its core some of the most basic and generally agreed functionalities of a bootloader are:

- Minimal hardware initialization. Especially since it's the first code the CPU executes upon power up. It might include:
  - Enabling access to / initializing internal RAM
  - Default initialization of MCU system clock for provision of timebase and prescalers.
  - Setting up Phase Lock Loops (PLL's)
  - Initialization of base addresses for Interrupt and Exception Trap Vector

  - Initialization of user stack pointer
  - Initialization of cache and/or external memory if such memory types are supported by the MCU.
- Identification of type of reset events (software, manual, hardware, power-up, via communication interface, etc.)
- Copy image from Flash or ROM to RAM for faster execution.
- Jump to application (alternatively to OS) and pass program control to it.

In the book *Real-Time Concepts for Embedded Systems* [1] for example, a typical flow is shown for a bootloader that is very similar to the one previously described. Such flow can be appreciated in Figure 1.

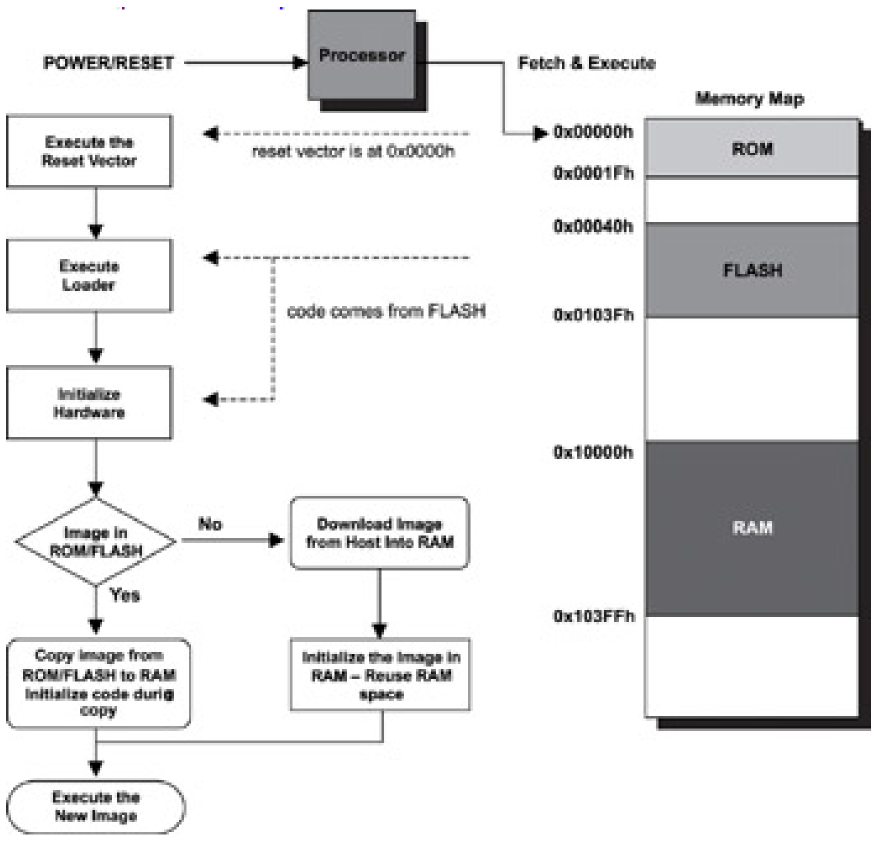


Figure 1 - Example of bootstrap overview [1]

Because of its key role, the bootloader usually occupies special boot blocks in the flash ROM, which have hardware protection against accidental erasure and corruption.

When it comes to the features considered as convenient to have in a bootloader, the following are amongst the most important:

- Checksum verification of application code
- Remote boot capability
- Reception of new application code (via a communication interface)
- Executing flash reprogramming routines from RAM
- Reception of a new flash image

In this regard, the authors of *Best Practices in Boot Loader Design* [2] consider a "good trait" to have the ability to perform network boot to quickly download a firmware image to the target device over a network using standard Internet protocols.

A similar capability is proposed in *Real-Time Concepts for Embedded Systems* [1] as it describes the whole concept of having a loader on the target side (embedded system) to download an image into RAM from a host system via a serial connection or even Ethernet, after completing the necessary initialization work.

2.2 Controller Area Network

The Controller Area Network (CAN) is a serial communication bus protocol developed by Robert Bosch GmbH and was first released in 1986. [3] The use of CAN have grown rapidly since the beginning of the 90's due to the increased number of ECU's used in cars and other vehicles. Today CAN is one of the most widely used communication bus system in the automotive industry. It is one of five protocols that are used in On-board diagnostics (OBD-II) and all cars and light trucks models 2008 and onward sold in the USA have CAN mandatory for diagnostic purpose. [4]

2.2.1 Concept of CAN

The CAN protocol covers two layers in the ISO/OSI Reference Model, Physical Layer and Data Link Layer, all Layers above is implementation specific. The Data Link Layer consists of two sublayers, Logic Link Control sublayer, which for example decides which messages that are received are accepted, and Medium Access Control sublayer, which is controlling Framing, Arbitration, Error Checking, Error Signaling and Fault Confinement. The Physical layer handles the actual transfer of bits between the nodes with respect to all electrical properties. [5]

A CAN network consists of nodes which are identified by an id number in the arbitration field. The arbitration field is used for prioritization of messages when multiple nodes want to send messages at the same time. The message with most dominant bits in the arbitration field has highest priority and will be transmitted without interrupt. CAN is specified to transmit data with a bit rate up to 1 Mbit/s in a network with a length below 40 m. The speed has to be decreased when long distance networks are used. [5]

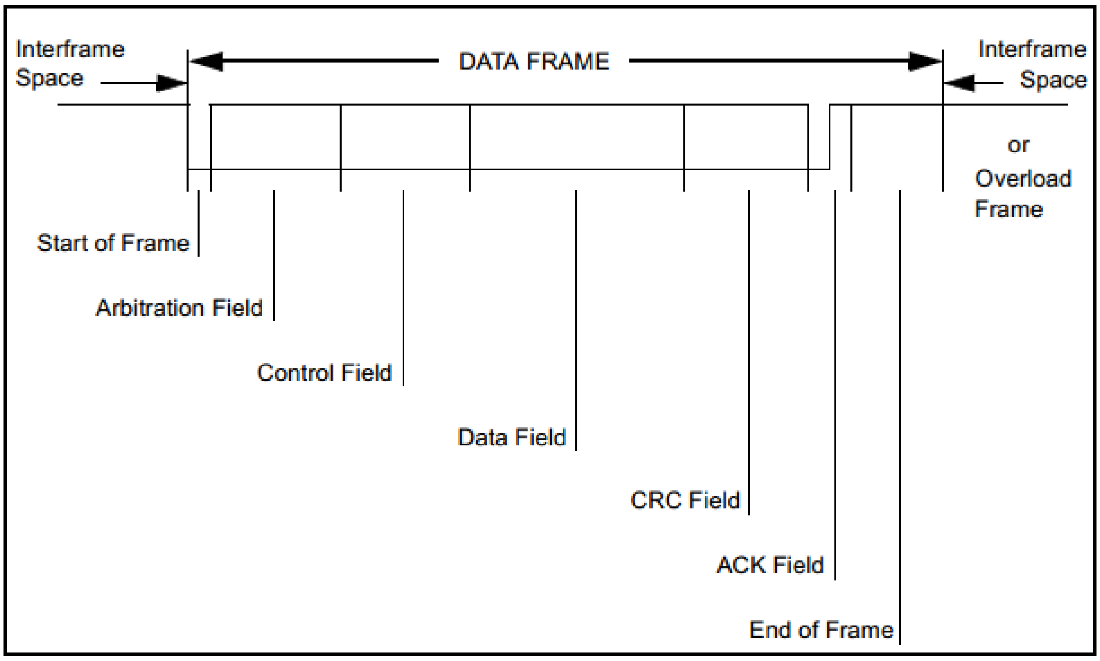


Figure 2 - CAN frame layout [5]

2.2.1.1 CAN frames

A normal CAN message frame consists of a 1-bit Start Frame, 11 or 29-bit Arbitration Field also called identifier, a 6-bit Control Field, a 0 to 8 byte long Data Field, a 16-bit CRC Field, a 2-bit Ack Field and a 7-bit End of Frame, seven recessive bits. CAN also have Remote Frames, which can be used to ask a node to send data, Error Frames , which are used to indicate error on the bus, and Overload Frame, which is used to inject a delay between messages.

2.3 AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) is a partnership between different OEM manufacturers and Tier 1 suppliers in the automotive industry that are working together to develop and establish an open industry standard for the automotive electrical and electronic architecture. The cooperation for an open system architecture begun in august 2002 when BMW, Bosch, Continental, DaimlerChrysler and Volkswagen started to discuss common challenges and objectives in the automotive industry. In November the same year a joint technical team was put together to establish strategies for the technical implementation and in May 2003 the partnership between the core partners was formally signed and AUTOSAR was born. In May 2010 the AUTOSAR cooperation consisted of 9 core members, 39 premium members, 11 development members, 57 associated members and 5 attendees. [6]

The basic goals of AUTOSAR are to be able to reuse software components so the rise in complexity of future automotive systems can be handled easily as well as aiming for optimization to be done at system level instead of component level. The benefit of having a standardized interfaces are the high degree of reuse of software and the ability to change to a different solutions from different suppliers but still have the same functionality. This reflects in the working principle of AUTOSAR, "Cooperate on standards, compete on implementation". Goals that are set up by the AUTOSAR development cooperation are [7]:

- Implementation and standardization of basic functions as OEM "Standard Core" solution
- Scalability to different vehicle and platform variants
- Transferability of functions throughout network
- Integration of functional modules from multiple suppliers
- Consideration of availability and safety requirements
- Redundancy activation
- Maintainability throughout the whole "Product Life Cycle"
- Increased use of "Commercial off the shelf hardware"
- Software updates and upgrades over vehicle lifetime

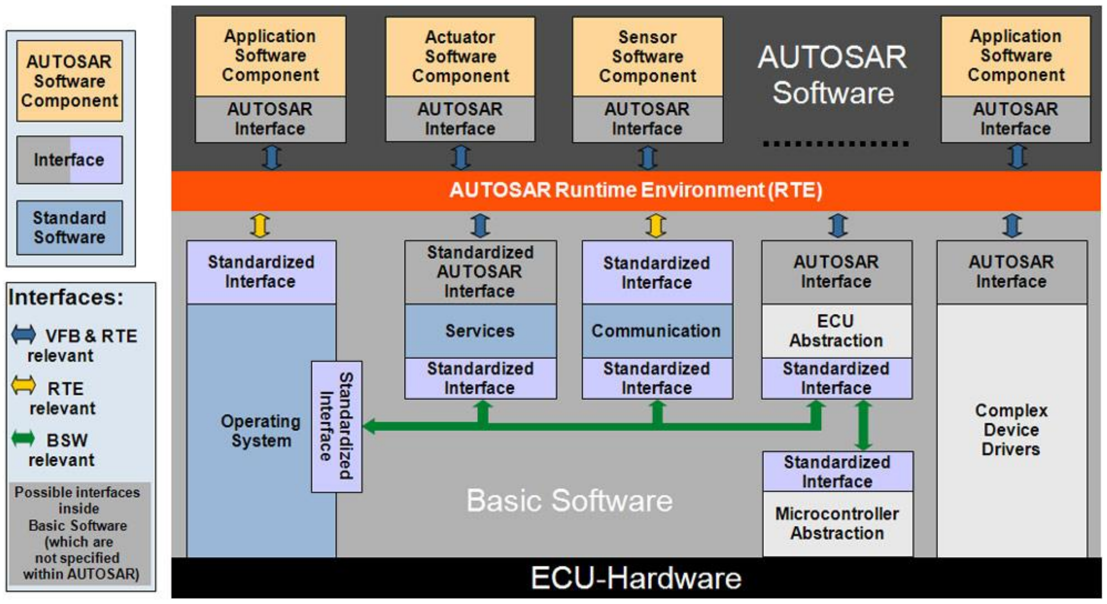


Figure 3 - Overview of AUTOSAR [8]

Figure 3 shows how AUTOSAR intends to standardize the interface between its basic infrastructure software modules, the hardware at low level and the software components at high level (application).

AUTOSAR is arranged into three main working topics, Software architecture, Methodology and Application interfaces.

Software architecture

The software architecture includes specifications for communications stacks, system services, diagnostic services, memory stacks, peripherals, implementation integration and real-time environments. This is all called AUTOSAR Basic Software.

Methodology

The methodology part of AUTOSAR aims to make the configuration process between the basic software stack and the application software in the ECUs as seamless as possible by defining exchange formats and description templates.

Application interfaces

The application interfaces topic includes specifications of typical automotive application interfaces in terms of syntax and semantics. This should serve as a standard for application software developed for AUTOSAR.

The architecture of AUTOSAR is built into three main software layers: the Application Layer, the Runtime Environment (RTE) Layer and the Basic Software (BSW) Layer which run in a microcontroller. [8]

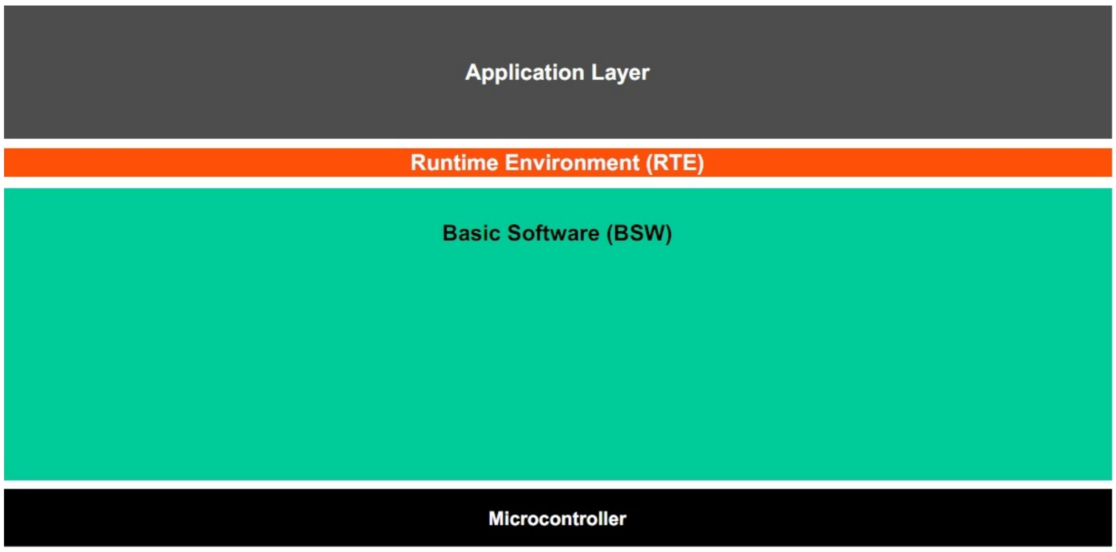


Figure 4 - Main software layers in the AUTOSAR architecture [8]

The Basic Software (BSW) main layer is further divided into the layers: Services, ECU Abstraction, Microcontroller Abstraction and Complex Drivers. The following image shows such division.

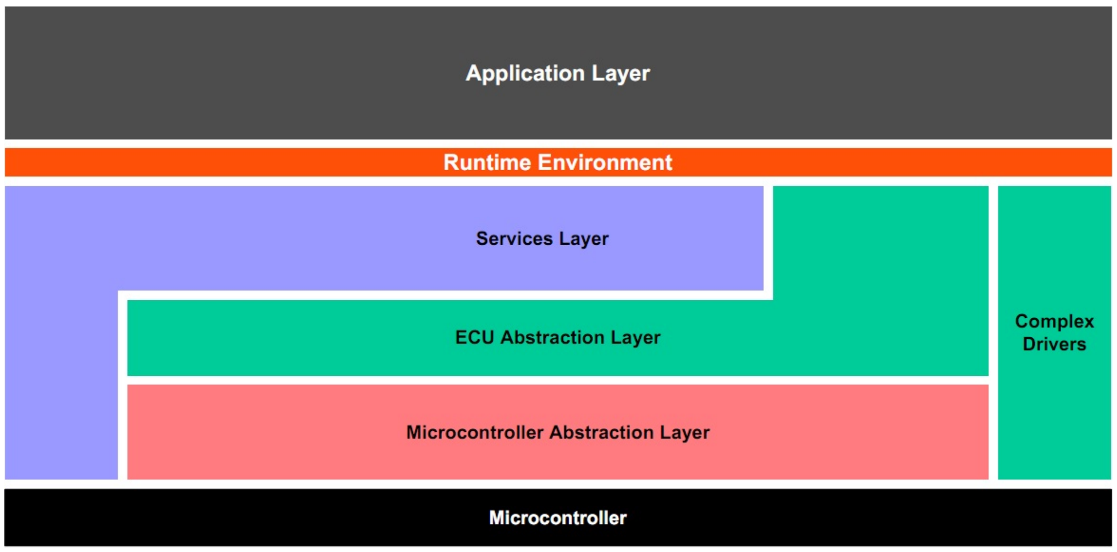


Figure 5 - Layer Division of the BSW Layer [8]

Figure 6 shows in what layer the modules are placed.

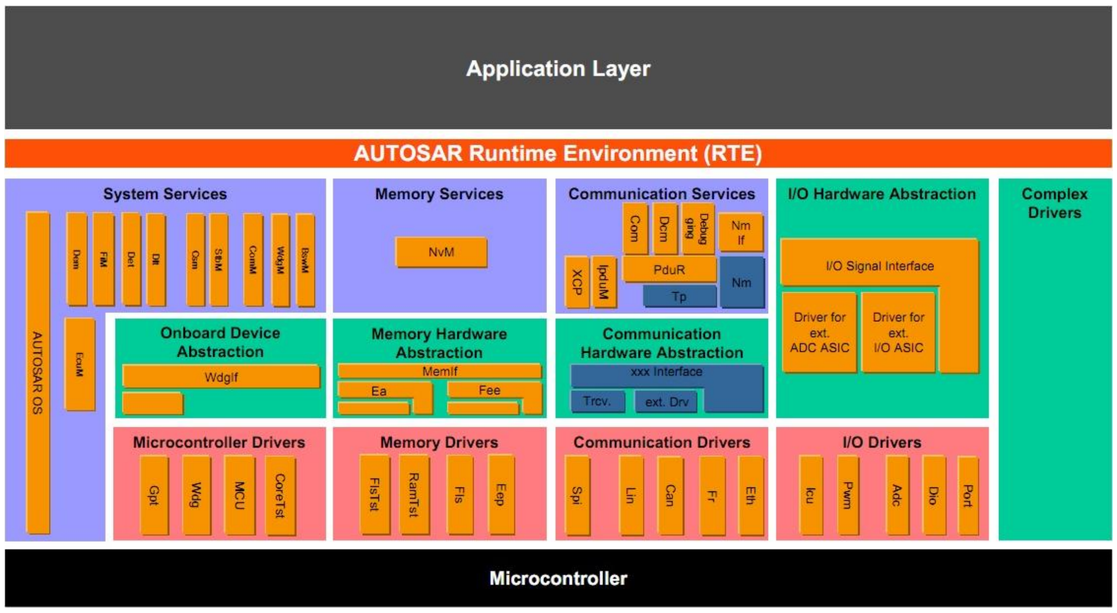


Figure 6 - The modules that resides in the layers [8]

2.4 Unified Diagnostic Services (UDS)

As defined by the International Standard Organization in the ISO 14229-1 standard ("1"= part 1), UDS is an automotive communications systems standard that specifies **data link independent** requirements of **diagnostic services**. To achieve this, the standard has been based on the Open System Interconnection (OSI) Basic Reference Model in accordance with ISO 7498-1 and ISO/IEC 10731, which structures communication systems into seven layers. When mapped on this model, the services used by a client and a server are divided as it follows:

- Unified Diagnostic Services (Layer 7)
- Communication Services (Layers 1-6)

In Table 1 it is possible to see the mapping between the OSI layers and the corresponding standards that specify services for each layer in an example implementation of diagnostics.

OSI Layer	**Enhanced Diagnostics Services**
Application (Layer 7)	ISO 14229-1/ISO 15765-3/ISO 11992-4
Presentation (Layer 6)	----
Session (Layer 5)	ISO 15765-3/ISO 11992-4
Transport (Layer 4)	ISO 15765-2/ISO 11992-4
Network (Layer 3)	ISO 15765-2/ISO 11992-4
Data Link (Layer 2)	ISO 11898/ISO 1992-1/SAE J1939-15
Physical (Layer 1)	ISO 11898/ISO 1992-1/SAE J1939-15

Table 1: Diagnostic specifications applicable to the OSI layers

Following the previous reasoning, application layer services are then usually referred to as diagnostic services. The application layer services are used in client-server-based systems. According to ISO 14229-1, a diagnostic service is in detail "an information exchange initiated by a **client** in order to require diagnostic information from a **server** and/or to modify its behavior for diagnostic purposes". The services are described independently of the communication protocols which will deliver them, implemented in lower layers.

Those services allow a tester (*client*) to control diagnostic functions in an on-vehicle Electronic Control Unit (*server*) applied for example on electronic fuel injection, automatic gear box, anti-lock braking system etc., connected on a serial data link embedded in a road vehicle. Furthermore, this part of the standard specifies generic services which allow the diagnostic tester (client) to store or to resume non-diagnostic message transmission on the data link. However, the part 1 of the standard does not specify any implementation requirements. Figure 7 shows a general configuration of a client-server connection within a vehicle network.

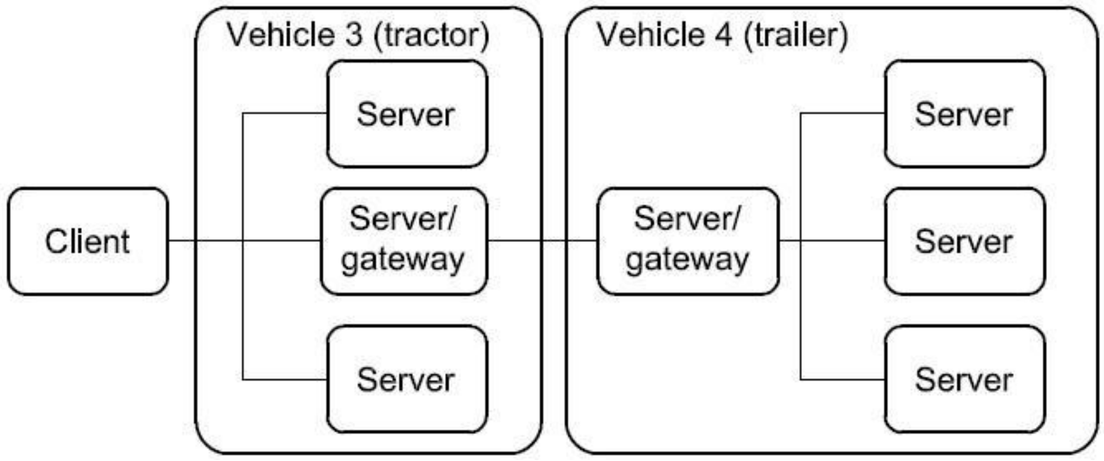


Figure 7 - UDS network [9]

For vehicle 3, the servers are directly connected to the diagnostic data link, and vehicle 4 connects its server/gateway directly to the vehicle 3 server/gateway. For vehicle 4, the servers are connected over an internal data link and indirectly connected to the diagnostic data link through the gateways. ISO 14229-1 applies to the diagnostic communications over the diagnostic data link; the diagnostic communications over the internal data link may conform to the same or to another protocol.

The server, usually a function that is part of an ECU, uses the application layer services to send response data, provided by the requested diagnostic service back to the client. The client is usually referred to as an External Test Equipment when is off-board but can in some systems, also be an on-board tester. The usage of application layer services is independent from the client being an off-board or on-board tester. It is possible to have more than one client in the same vehicle system.

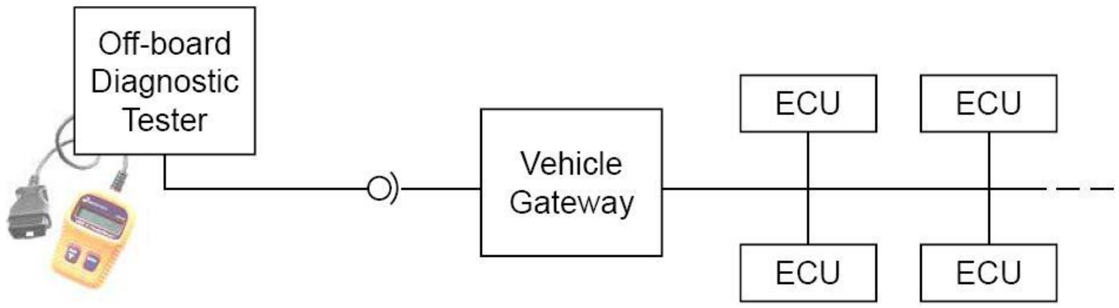


Figure 8 - The client as an Off-board tester.

Most typical network configuration of the client-server communication for vehicle diagnostics: the client as an Off-board tester, see Figure 8. Communication is based on a request-response model.

In the context of diagnostics, the following concepts are useful for a better understanding of the semantics handled on the UDS standard environment:

- **Diagnostic Trouble Codes (DTC)**: Numerical common identifier for a fault condition identified by the on-board diagnostic system.

- **Diagnostic Data**: Data that is located in the memory of an electronic control unit which may be inspected and/or possibly modified by the tester (diagnostic data includes analogue inputs and outputs, digital inputs and outputs, intermediate values and various status information). EXAMPLES: vehicle speed, throttle angle, mirror position, system status, etc.

- **Diagnostic Session**: Current mode of the server, which affects the level of diagnostic functionality.

- **Diagnostic Routine**: Routine that is embedded in an electronic control unit and that may be started by a server upon a request from the client. NOTE: It could either run instead of a normal operating program or run concurrently to the normal operating program. In the first case, normal operation of the ECU is not possible. In the second case, multiple diagnostic routines may be enabled that run while all other parts of the electronic control unit are functioning normally.

- **Tester**: System that controls functions such as test, inspection, monitoring or diagnosis of an on-vehicle electronic control unit and which may be dedicated to a specific type of operator (e.g. a scan tool dedicated to garage mechanics or a test tool dedicated to assembly plant agents)

As stated before, UDS is independent of lower layer protocols. Therefore, car diagnostics could be executed, for example, by using the Unified Diagnostic Services (UDS) on top at an application layer level, generic network layer services at an intermediate level and employing the means of controller area networks (CAN) or another communication bus protocol (like LIN, FlexRay or Ethernet) for the data link and physical layers' levels, at lower stages. Such approach is compliant with the generic OSI model for networks. Figure 9 shows the hierarchical arrangement of layers for the implementation of diagnostic services over CAN according to standard ISO-15765-3.

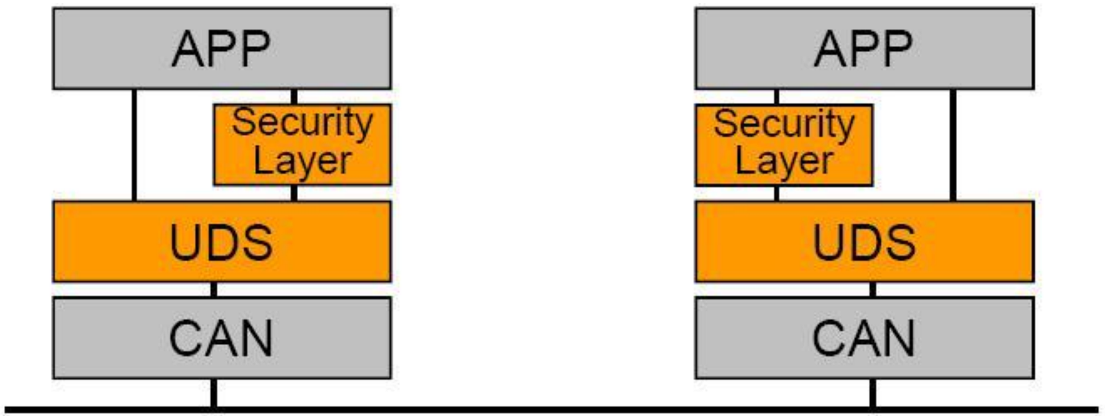


Figure 9 - UDS also defines a security layer in order to encrypt data.

When executing diagnostics over CAN, the Client (diagnostic tester) initiates a request and waits for confirmation. The server (function in ECU) then receives the indication and sends response.

Besides specifying services' primitives and protocols that describe the client-server interaction, UDS also defines within its framework a number of functional units that comprise several services each, identified with a hexadecimal code. These units are intended for different individual purposes that support the overall diagnostic function/task. Such units are:

- Diagnostic and Communication Management
- Data Transmission
- Stored Data Transmission
- Input/Output Control
- Remote Activation of Routine
- Upload/Download

The following table shows specific examples of use cases for diagnostics, some of the services that could be employed for such purposes and the corresponding functional units that implement them:

Use Case	**Service(s)**	**Functional Unit**
Observation of data stored within a system (e.g. trouble codes or some form of identifications)	ReadDTCInformation	Stored Data Transmission
Observation of live data (such as engine or vehicle speed)	ReadDataByIdentifier, ReadDataByPeriodicIdentifier	Data Transmission
Transfer of large amount of data (for example, reflashing a module)	RequestDownload, TransferData, ECUreset, DiagnosticSessionControl	Upload/Download, Data Communication Manager
Control of a module's I/O (for example, disabling individual cylinders to identify a fault)	InputOutputControlByIdentifier	Input/Output Control
Run of specific routines already in a module (like some sort of in-built self calibration)	RoutineControl	Remote Activation Of Routine

Table 2 - Services provided by different functional units defined in the UDS standard applied to their respective use cases.

2.4.1 Relationship between UDS and AUTOSAR

The Diagnostic Communication Manager (DCM) module within the AUTOSAR architecture provides a common API for diagnostic services. The functionality of the DCM module is used by external diagnostic tools during the development, manufacturing or service.

The DCM module ensures diagnostic data flow and manages the diagnostic states, especially diagnostic sessions and security states. Furthermore, the DCM module checks if the diagnostic service request is supported and if the service may be executed in the current session according to the diagnostic states.

There's a close relationship between the services established in the UDS standard and the DCM. The DCM module provides the OSI layers 5 to 7 of an implementation of diagnostics. At OSI-level 7, the DCM provides an extensive set of ISO 14229-1 services (21 of 25 possible). At OSI-level 5, the DCM module handles the network-independent sections of the following specifications:

- ISO 15765-3: Implementation of unified diagnostic services (UDS over CAN).
- ISO 15765-4: Requirements for emission-related systems.

Table 3 provides a view of such layered structure:

OSI Layer	**Protocols**
7: Application	UDS ISO 14229-1 / Legislated OBD ISO 15031-5
6: Presentation	---
5: Session	ISO 15765-3
4: Transport	ISO 15765-2
3: Network	ISO 15765-2
2: Data Link	CAN / LIN / FlexRay / MOST
1: Physical	CAN / LIN / FlexRay / MOST

Table 3 - OSI model

This arrangement matches the mapping of layers established by the UDS standard in the ISO 14229-1 specification, which was also shown before. In addition, to remark the support of AUTOSAR to UDS, it will be pointed later where are the UDS services included within the overall functionalities provided by the DCM module. [10]

In the AUTOSAR architecture, the Diagnostic Communication Manager is located in the Communication Services (Service Layer), see Figure 10.

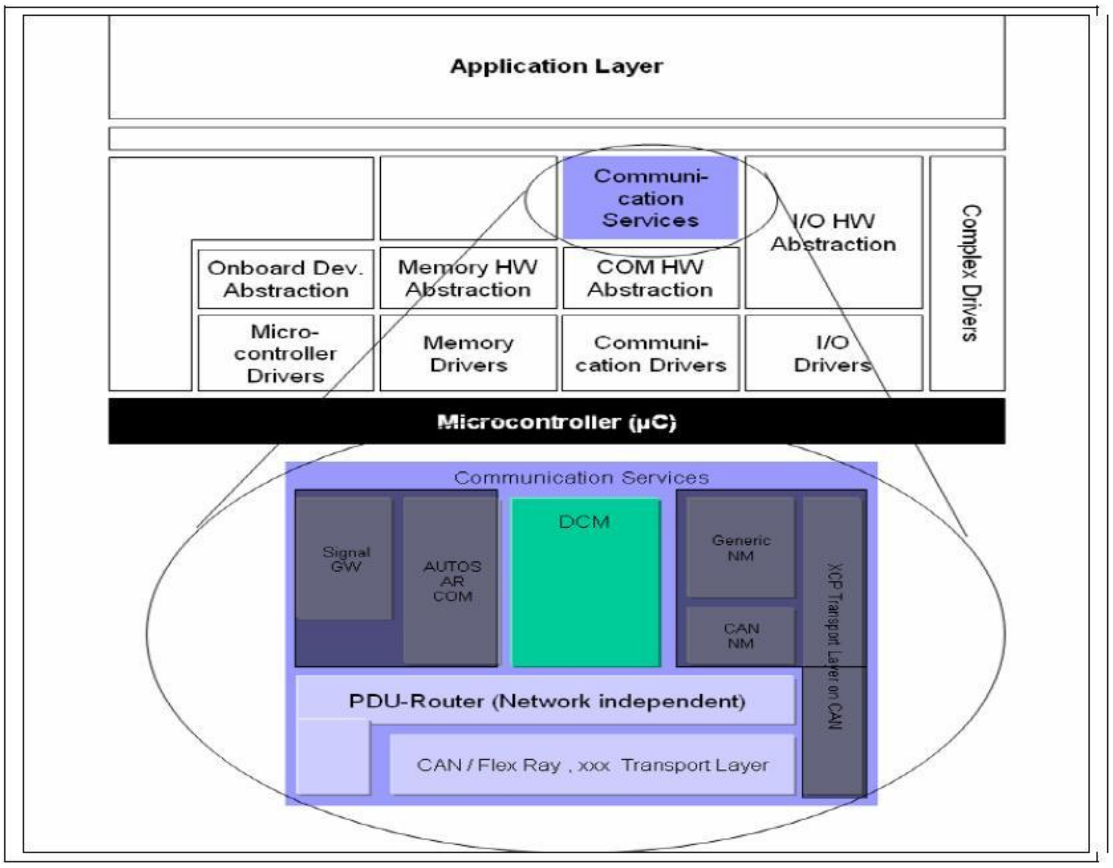


Figure 10 - Position of the DCM module in the AUTOSAR architecture [10]

The DCM module has the capability of being network-independent. All network-specific functionality (the specifics of networks like CAN, LIN, FlexRay or MOST) is handled outside of the DCM module. The Protocol Data Unit Router (PduR) module provides a network-independent interface to the DCM module.

The DCM module receives a diagnostic message from the PduR module. The DCM module processes and checks internally the diagnostic message. As part of processing the requested diagnostic service, the DCM will interact with other BSW modules or with SW-components (through the RTE) to obtain requested data or to execute requested commands. This processing is very service specific. Typically, the DCM will assemble the gathered information and send a message back through the PduR module. [10]

Dependencies to other modules

- *Diagnostic Event Manager (DEM)*: The DEM module provides function to retrieve all information related to fault memory such that the DCM module is able to respond to tester requests by reading data from the fault memory. [11]

- *Protocol Data Unit Router (PduR module)*: The PduR module provides functions to transmit and receive diagnostic data. Specifically, it provides DCM with data of incoming diagnostic requests. [12] Proper operation of the DCM module presumes that the PduR interface supports all service primitives defined for the Service Access Point (SAP) between diagnostic application layer and underlying transport layer. [9]

- *Communication Manager (ComM)*: The ComM module provides functions such that the DCM module can indicate the states "active" and "inactive" for diagnostic communication. The DCM module provides functionality to handle the communication requirements "Full-/ Silent-/ No-Communication". Additionally, the DCM module provides the functionality to enable and disable Diagnostic Communication if requested by the ComM module. [13]

- *SW-C and RTE*: The DCM module has the capability to analyze the received diagnostic request data stream and handles all functionalities related to diagnostic communication such as protocol handling and timing. Based on the analysis of the request data stream the DCM module assembles the response data stream and delegates routines or IO-Control executions to SW-Cs .If any of the data elements or functional states cannot be provided by the DCM module itself the DCM requests data or functional states from SW-Cs via port-interfaces or from other BSW modules through direct function-calls. [14]

- *BswM*: The BswM allows the DCM to request a mode change, e.g. in case of reset or session change

In Figure 11, the dependencies between the DCM module and other AUTOSAR modules are visible.

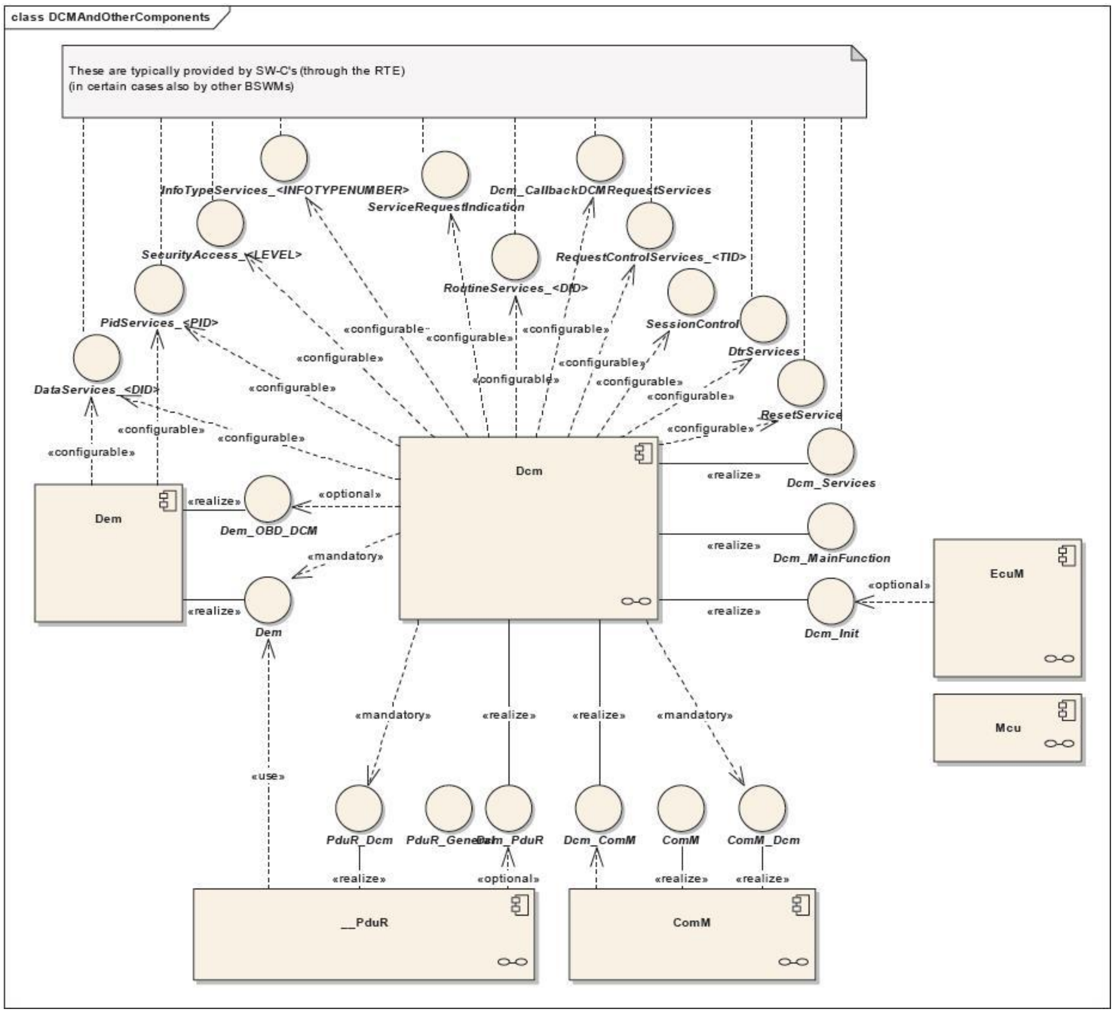


Figure 11 - Interaction between DCM and other modules [10]

A whole Application Programming Interface (API) for the DCM module is defined in *AUTOSAR Specification of Diagnostic Communication Manager* [10] and is used for handling

a) The syntax and semantics of the functions that are provided and required form other Basic Software (BSW) modules. These take the form of API's reminiscent to C language.
b) The syntax and semantics of a subset of those functions which are used by software components through the Run Time Environment (RTE).

In addition to the previously mentioned links between AUTOSAR and UDS, many data types and functions within the DCM's API show more bonds by means of the following features:

- Borrow/loan values and services that are specified or used by UDS.

- Provide results and define further services to those required by UDS.

2.4.2 An UDS use case: Flash Reprogramming

The purpose of this section is to show the process and items that could be used in terms of data transfer between a client and a server, as suggested by UDS.

One of the uses of UDS is the reprogramming of flash memory in an ECU, typically to load a new version of an application program with corrections (fix) or enhancements (upgrade).

The main job of UDS in such operation would be:

- To set the server into a reprogramming session mode
- To start a reprogramming sequence
- To handle the start and stop of data transfer
- To handle the size and right order of both the data blocks being sent/received and the memory blocks where such data blocks should be stored.
- To allow the client to start/stop a routine that runs on the server
- To allow the client to request a software reset event on the server

Despite the fact that AUTOSAR does not explicitly specify, and thus still does not fully support the process of flash reprogrammming as stated in *AUTOSAR Specification of Flash Driver* [15], the newest AUTOSAR version 4.0 has just included in the DCM the support of some crucial UDS services that can be applied for taking care of most of the reprogramming-related actions written above:

- ReadMemoryByAddress
- WriteMemoryByAddress
- RequestDownload
- RequestUpload
- TransferData
- RequestTransferExit
- CommunicationControl
- ResponseOnEvent

It can be seen then that the AUTOSAR specification relies strongly on UDS to perform flash reprogramming.

2.5 Universal Measurement and Calibration Protocol (XCP)

The Universal Mesurement and Calibration Protocol (XCP) is a ASAM, “Association for Standardization of Automation and Measuring Systems”, standardized protocol primarily used for calibration and measurement of parameter data in ECUs.
In the 1990s the CAN Calibration Protocol (CCP) was developed as a tool to enable flexible read and write access to variables in a ECU. CCP only supported CAN communication because it was the dominant bus to be used in automotive at that time. More bus systems like LIN, FlexRay and Ethernet was introduced in the automotive industry and CCP became limited in used because it could only support the CAN interface and no other. This led to the development of the Universal Calibration Protocol, XCP, in 2003.

XCP is constructed in two layers, a transport layer and a protocol layer and is a Single-Master Multi-Slave system. The protocol layer is interface independent and describes the architecture, syntax and settings of the protocol. The protocol layer is the same whatever communication interface that is used. The transport layer specifies how XCP should transport data over the network and changes depending on what communication interface is used.
In May 2010 the ASAM XCP standard supported XCP on CAN, SPI, SCI, TCP/IP, UDP/IP, USB and FlexRay.

The XCP Single-Master is the measurement and calibration system, often a PC, and the ECUs operates as Slaves. The master establishes a communication channel, a point-to-point connection, with the slave. The connection must be point-to-point between master and slave and broadcasting XCP messages is not allowed, except XCP message GET_SLAVE_ID, which can be broadcasted. Multiple masters are not allowed but a master can have multiple communication channels active in the same network at the same time.

Every slave in the system have a ECU description file, A2L-file, which specifies what checksum method that is used, the physical meaning of the data, the link between variable names and their address range and so on. The XCP master can access these descriptions files and read out all the necessary information it might need.

The communication between master and slave is done through the XCP driver that is integrated in the master. In XCP Standard communication mode each request packet send form the master, the slave will respond with a response packet or an error packet. The master will not send a new packet until it receives a response or error packet from the slave regarding previous sent request.
To speed up the process of uploading and downloading to memory and programming the flash, Block Transfer Communication Mode can be applied. This communication mode is similar to the one that is used in UDS. In this mode multiple requests are sent at the same time as a block of data and the slave responds by sending a block of responses back to the master.

Another way to enhance transfer speed is to use the Interleaved Communication Mode where the master doesn't wait for a response packet until it sends a new request. Number of requests that can be sent interleaved by the master depends on the queue size of the slave. Block Transfer Communication Mode and Interleaved Communication Mode cannot be used in the same system.

A XCP message consists of three parts, Header, Packet and Tail, see Figure 12. The Header and the Tail is part of the Transport layer and consist of a Control Field which content depends on what bus is used. The Packet part of a XCP message consists of an Identification Field, Timestamp Field (optional) and a Data Field and is a generic part of the protocol layer.

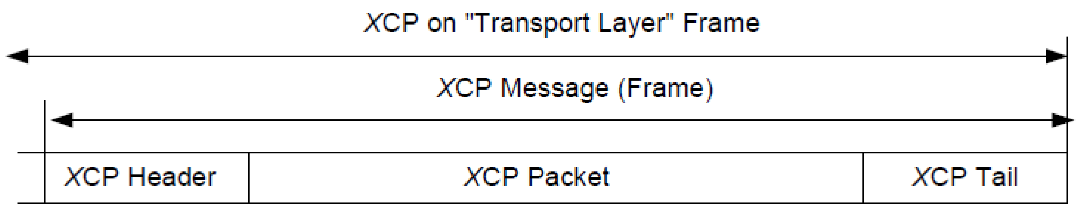


Figure 12 - XCP frame

2.5.1 Packets

Two different types of packets are used in the communication via XCP, Command Transfer Object (CTO) and Data Transfer Object (DTO).
The CTO have five types of objects:

- CMD (Command)
- RES (Response)
- ERR (Error)
- EV (Event)
- SERV (Service Request Processor)

The DTO have two types of objects that are both used for event driven reading of variables from, or writing values to, the memory of the slave.

- DAQ (Data Acquisition)
- STIM (Stimulation)

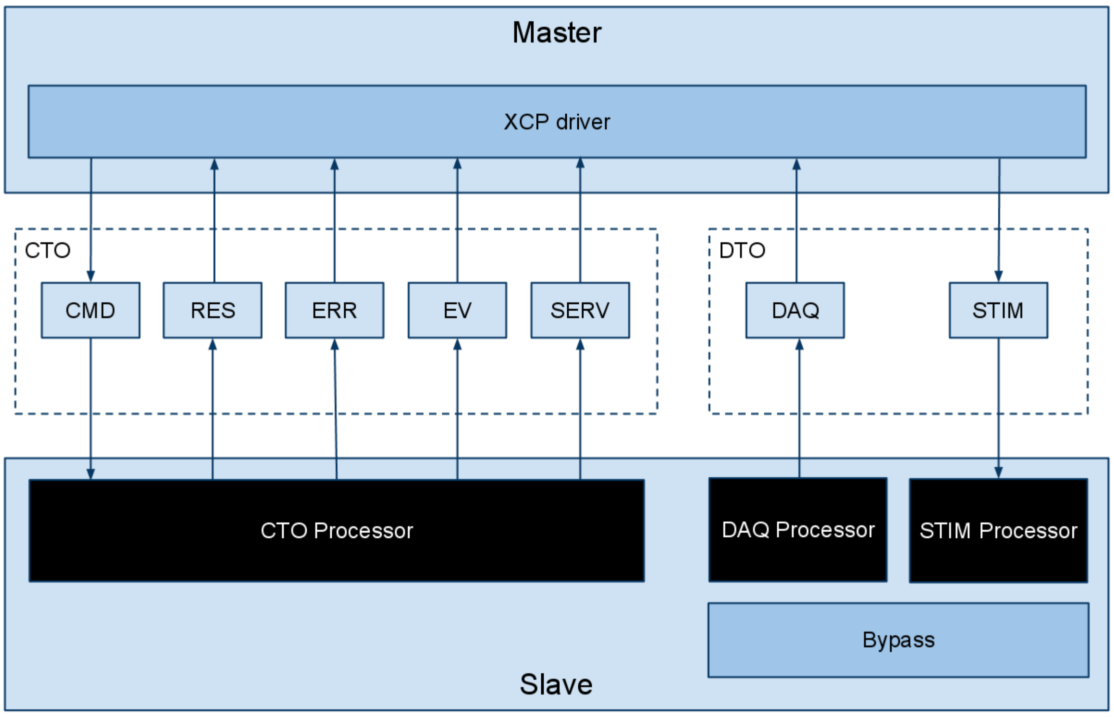


Figure 13 - Communication objects between master and slave

2.5.2 Security

To ensure that access to the ECU is restricted to authorized persons only a security mechanism called Seed & Key is a part of the XCP specification. A slave unique key is needed to grant access to a slave and the master must ask the slave for a seed to be able to compute the key and gain access. The algorithm to calculate the key is unknown to the master to ensure confidentiality. The key algorithm is encapsulated in a file called SeedNKey.DLL which the master uses to calculate the key with help of the seed provided by the slave.

2.5.3 Flash programming

In XCP the object called SECTORS describes the physical layout of the ECU memory. The start addresses and sizes of the SECTOR are of high important when reprogramming the ECU.
Programming of flash memory in a slave device can be divided into three parts.

1. **Administration before programming**. For example to check what version of the software is currently loaded in the memory.
2. **The flash process**. This is when the programming of the memory takes place.
3. **Administration below**. For example to do a checksum control to see if the flashing process was successful.

The XCP standard does not support special commands for version control because the administration steps are very project specific and depends on the ECU, but the ECU functional description can specify which standard XCP commands can be used for administration actions like version control.

The XCP standard offers seven commands to be used special for programming:

- PROGRAM_START
- PROGRAM_CLEAR
- PROGRAM_FORMAT
- PROGRAM
- PROGRAM_VERIFY
- PROGRAM_RESET

How these commands are used in a project must be specified in a project specific "programming flow control" document. [16]

2.5.4 Flash memory access

There are two different flash access methods supported by the XCP protocol to be used in the flash process. Absolute access mode and Functional access mode which both uses the same commands but with some different parameters.

Absolute Access Mode, access by address

This is the default mode and is used when the physical layout of the flash memory is well known to the programming tool. The flash content to be programmed must be available and the address information of the data must be known. The physical layout information can be read out of the ECU or in a description file depending on how it is done in the project. The block of data is contained in the Command Transfer Object (CTO) will be programmed into the flash memory starting at the Memory Transfer Address (MTA) which will be incremented by the number of bytes of data that are being programmed into the memory. [17]

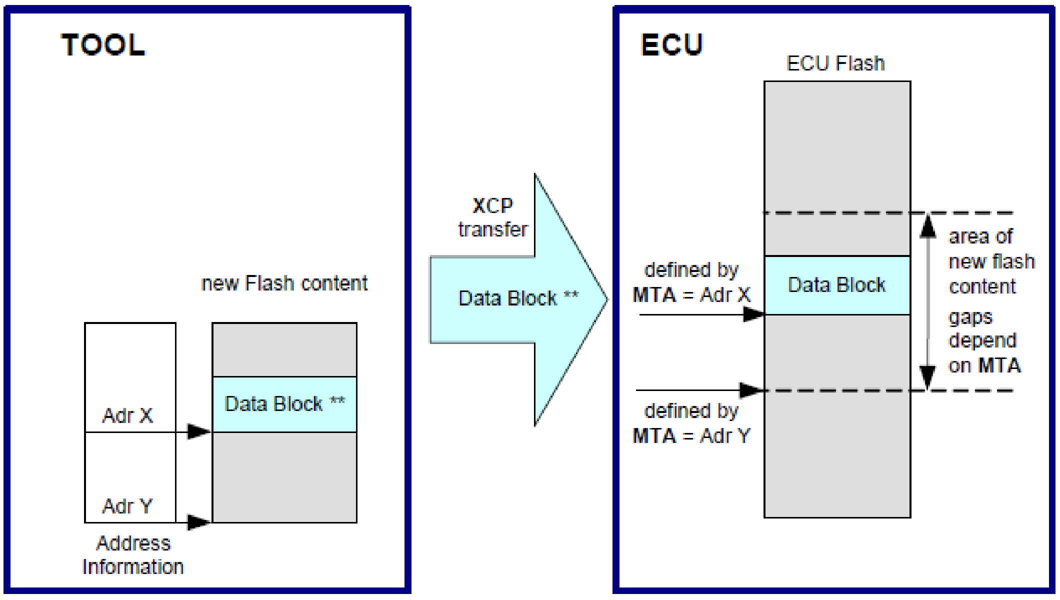


Figure 14 - Absolute Access Mode in XCP

Functional Access Mode, access by flash area

This behavior is similar to how UDS, ISO 14229-1 and ISO 15765-3, works.

In Functional Access Mode the tool does not need to to know any information about the memory mapping or address of the flash content to be programmed. The only information the tool needs is the area of the flash. A pointer pointing on the data block represents the address information.

The block of data contained in the CTO will be programmed into the flash memory and the start address for the new content is automatically known by the ECU software. The MTA is counted inside the master and the server and is working as a Block Sequence Counter (BSC) that allows an improved error handling. This is a good function if the programming service fails during a sequence of multiple programming requests. The BSC in the server shall be initialized to a 1 when it receives the PROGRAM_FORMAT request message so the first PROGRAM request message starts with a BSC of 1. The BSC is incremented by 1 for every subsequent data transfer request. When the BSC reaches it maximum value it rolls over and starts over at 0X00.

If the PROGRAM request is not received by the slave or if the positive response message from the slave is not received by the master the master will wait for a timeout and repeat the same request including the same BSC. The slave will then write the data into the memory and send a positive response message or just send a positive response message if it already has received that PROGRAM request before.

After when the flash process is finished a checksum control to verify the data can be performed with the BUILD_CHECKSUM and PROGRAM_VERIFY commands. The end of a programming session is indicated by sending a PROGRAM_RESET command and the slave will then go into a disconnect state and often a hardware reset of the slave is executed after that. [18]

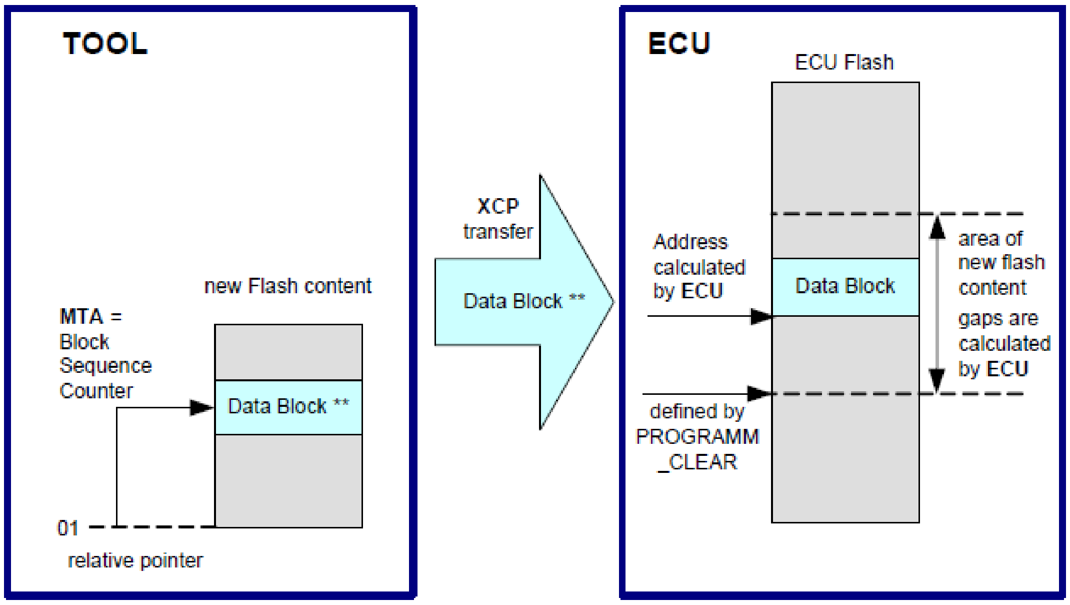

TOOL
new Flash content
MTA = Block Sequence Counter
Data Block **
01
relative pointer
XCP transfer
Data Block **
ECU
ECU Flash
Address calculated by ECU
Data Block
defined by PROGRAMM _CLEAR
area of new flash content
gaps are calculated by ECU

Figure 15 - Functional Access Mode in XCP

2.5.5 Relationship between XCP and AUTOSAR

Dependencies of other AUTOSAR modules

BSW Scheduler
Calls the main function of XCP, which are necessary for cyclic processes.

FlexRay Interface
To be able to transmit and receive XCP PDUs via FlexRay the AUTOSAR FlexRay Interface module is needed to be used.

CAN Interface
To be able to transmit and receive XCP PDUs via CAN the AUTOSAR CAN Interface module is needed to be used. [19]

SocketAdaptor
To be able to transmit and receive XCP PDUs via Ethernet the AUTOSAR SocketAdapter module is needed to be used.

RTE
To copy calibration parameters from ROM/FLASH and to use the double pointered method the RTE is used. [14]

OS
The time stamped feature of XCP uses the AUTOSAR OS counter.

Diagnostic Event Manager
The DEM module provides function to retrieve all information related to fault memory such that the DCM module is able to respond to tester requests by reading data from the fault memory. [11]

Development Error Tracer
XCP has to have access to the error hook of the Development Error Tracer to be able to report development errors.

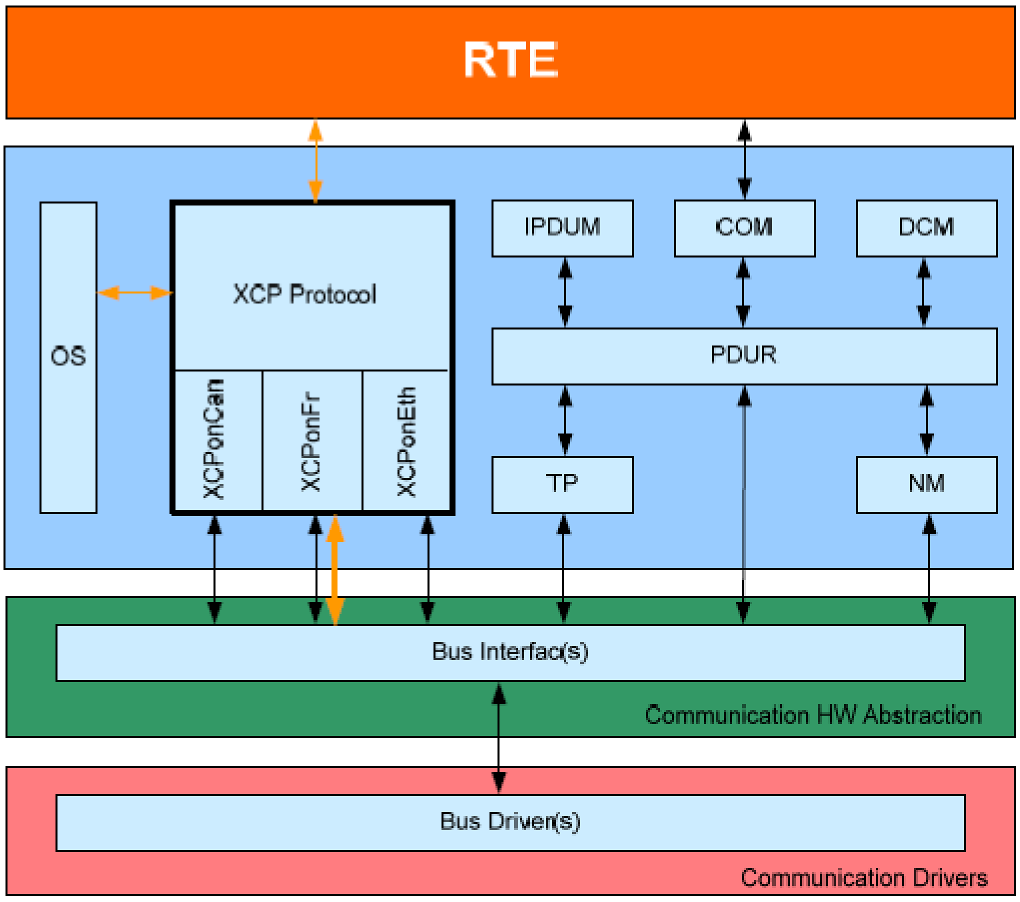


Figure 16 - XCP in AUTOSAR

AUTOSAR supports XCP on CAN, FlexRay and Ethernet and the following features of XCP version 1.1:

- A2L IF_DATA Section Support
- Synchronous data acquisition
- Synchronous data stimulation
- Block communication mode
- Interleaved communication mode
- Dynamic data transfer configuration
- Timestamped Data transfer
- Bypassing
- Seed & Key
- Online Calibration Data Page Switching
- DAQ configuration storing with power-up data transfer (RESUME mode)

The XCP feature of flash programming of the ECU is currently not a part of the AUTOSAR specification. The use of XCP or CCP is an alternative to reprogramming with the Flash Bootloader. This variant is primarily used to optimize ECU parameters. Since XCP/CCP programming is generally used in the development phase, unlike the Flash Bootloader, security aspects play a subordinate role here. [20]

2.6 ODEEP QR5567 Development platform

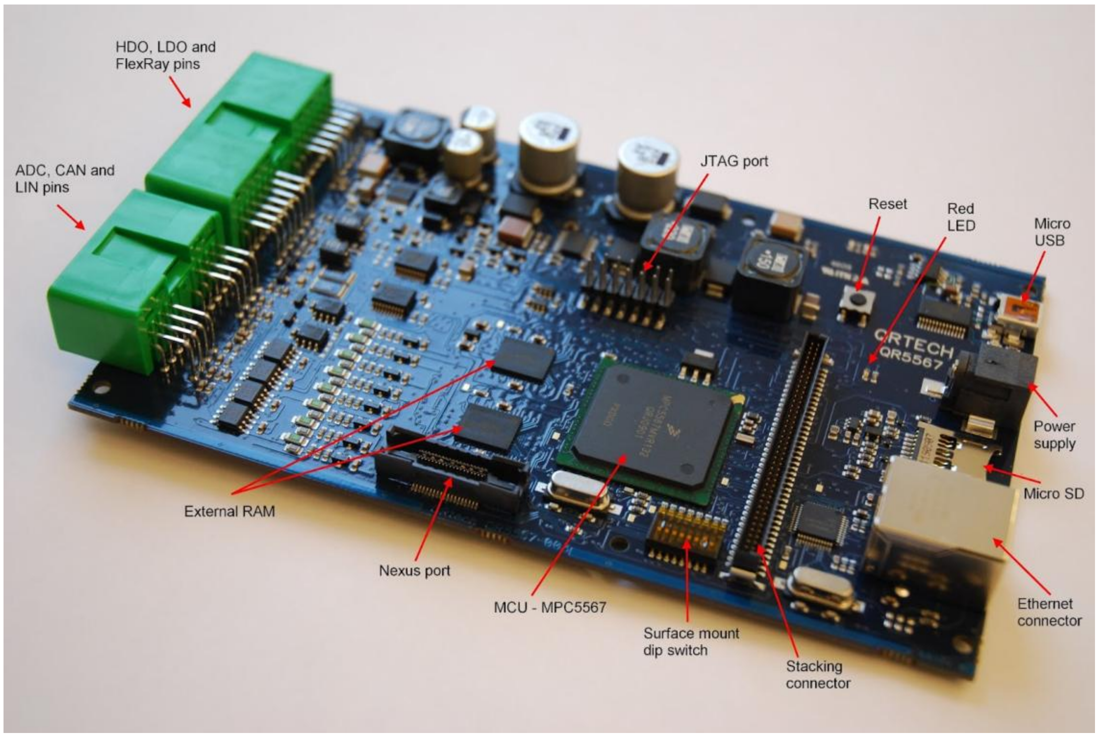


Figure 17 - ODEEP QR5567 Development platform

ODEEP QR5567 is a development platform made for rapid prototyping made by QRTECH AB. The platform is equipped with a 32-bit FreeScale MPC5567 microcontroller running at 128 MHz with 2 MB flash and 64 kB RAM memory built in. An additional 32 MB memory is also available on the platform. Except the Nexus and the JTAG ports for debugging and reprogramming the platform also has the following interfaces [21]:

- Four CAN 2.0B interfaces with TJA1050 tranceivers
- Two LIN 2.0 interfaces
- Two FlexRay interfaces with TJA1080 tranceivers
- 10/100mbit Ethernet interface
- Four 3.0A High Side Driver Outputs (HDO)
- Four 2.8A Low Side Driver Outputs (LDO)
- Eight analog or digital inputs
- Micro SD-Card Interface
- 5V External Output
- Two H-Bridge Drivers
- One Universal Serial Bus (USB) interface

2.6.1 MPC5567

The core of the ODEEP platform is the MPC5567 microcontroller produced by FreeScale Semiconductor, Inc. MCP5567 is a 32-bit RISC processor built on the Power Architecture™ with 80 KB internal SRAM and 2 MB flash memory built in. The microcontroller is specially developed for the automotive industry and has built in FlexRay and CAN controllers. The microcontroller is built around the e200z6 CPU with a maximum clock speed of 132 MHz.

Other important parts for reprogramming are the Boot Assist Module (BAM) which allows setting up different booting modes and the five FlexCAN modules that are implemented according to version 2.0B of the CAN communication protocol. Each FlexCAN module contains 64 message buffers for temporary storage of incoming and outgoing messages. [22]

The MCU has internal SRAM and FLASH and externally available RAM.

Description	**Address Space**	**Size**
SRAM	0x4000_0000h - 0x4001_3FFF	80 Kbytes
FLASH	0x0000_0000h - 0x001F_FFFFh	2 Mbytes
External RAM	0x2000_0000h - 0x21FF_FFFCh	32 Mbytes

3 Method and implementation

3.1 Bootloader functionality

3.1.1 Setup of MCU (Primary Bootloader)

As described in the Theoretical Background section, there is a part of code within the bootloader responsible for initializing and setting up the resources of the MCU so the part of code corresponding to an application can run properly later on. Additionally, this first part of code could handle a possible external prompt for reprogramming flash memory via a communication interface. In the context of this work such piece of code is identified as a Primary Bootloader (PBL).

The e200z6 core within the MPC5567 microcontroller makes use of the Boot Assist Module (BAM) and the status of the BOOTCFG pins to start the execution of code after reset. The BAM contains the MCU's most basic boot program code.

Boot Assist Module startup procedure

The Boot Assist Module (BAM) configures the Memory Management Unit (MMU) of the processor to allow access to all internal resources of the MCU and access to the external memory space as well. In the next step the BAM reads the status of the BOOTCFG pins in the reset status register (SIU_RST) and the defined boot sequence is started. Internal boot mode configuration is to be used in this project. [22]

BOOTCFG [0:1]	Censorship Control	Serial Boot Control	Boot Mode Name	Internal Flash State	JTAG State	Serial Password
00	**!0x55AA**	**Don't care**	**Internal-Censored**	**Enabled**	**Disabled**	**Flash**
00	**0x55AA**	**Don't care**	**Internal-Public**	**Enabled**	**Enabled**	**Public**
01	**Don't care**	**!0x55AA**	**Serial-Password**	**Enabled**	**Disabled**	**Flash**
01	**Don't care**	**0x55AA**	**Serial-Public Password**	**Disabled**	**Enabled**	**Public**
10	**!0x55AA**	**Don't care**	**External-No Arbitration-Censored**	**Disabled**	**Enabled**	**Public**
10	**0x55AA**	**Don't care**	**External-No Arbitration-Public**	**Enabled**	**Enabled**	**Public**
11	**-**	**-**	**Not Available**	**-**	**-**	**-**

Table 4 - Boot Modes

Note: !0x55AA means all other values except 0x55AA

Internal boot mode sequence starts with the BAM setting up a bus error exception handler in case of bus error due to accessing flash memory locations that may be corrupted. Next the BAM tries to find a valid Reset Configuration Half Word (RCHW) that is placed in six predefined locations in the internal flash memory.

A valid RCHW is a 16-bit value, 8-bit is boot identifier 3-bits are configuration bits and 5-bits are reserved and not used by RCHW. The first half word in one of the low address spaces flash blocks is expected to be the RCHW, see Table 5. If a valid RCHW is found the BAM enables the watchdog timer in the processor with the RCHW[WTE] bit. The timeout of the watchdog is 2.5x217 system clock periods.

Block	Address
0	0x0000_0000
1	0x0000_4000
2	0x0001_0000
3	0x0001_C000
4	0x0002_0000
5	0x0003_0000

Table 5 - RCHW locations

The BAM also fetches the reset vector from the address of the BOOT_BLOCK_ADDRESS + 0x4 and branches to the reset boot vector. At the reset boot vector address, valid user application instructions should be placed. If a valid RCHW is not found, the BAM will proceed to serial boot mode.

For this implementation a memory region was specifically defined for the RCHW in the linker script file. It was called “flash_rchw” and it starts at address 0x0000000 having a size of 8 bytes. A corresponding memory section identified as “.rchw” was assigned to this region. In the present work, since it was designed to boot from internal flash the boot ID of the RCHW must be read as 0x5A. This was achieved by setting the 4 bytes of that section to the boot ID value. The 4 bytes left are set to the address of the reset vector to which the BAM jumps i.e. the start address of the initialization part of the Primary Bootloader. [22]

Startup and initialization part

So then, after the BAM module has done the first initialization of the processor it will start running the code actually developed for the Primary Bootloader. Following are the actions in detail done by the startup and initialization part:

- Enable of Signal Processing Extension (SPE) instructions: This is done to allow the processor to operate entirely on all of the 64 bits of the General Purpos Registers (GPRs). The SPE defines load and store instructions for transferring 64 bit-values to/from memory.
- Initialization and start of global time base: To be able to count with system clock count
- Setup of Memory Management Unit (MMU): To provide memory translation from effective to real addresses. In this case to allow for flash reprogramming, effective addresses are the same as the real addresses. For the same purpose, the CPU must access memory mapped regions which is also configured here by setting up these MMU entries:
  - Entry 0: Peripheral Bridge B and BAM (1 Mbyte page size)
  - Entry 1: Internal Flash (16 Mbyte page size)
  - Entry 2: External Bus Interface for acces to External SRAM (16 Mbyte page size)
  - Entry 3: Internal SRAM (256 Kbyte page size)
  - Entry 4: Peripheral Bridge A (1 Mbyte page size)
- Initialization of Internal SRAM: To initialize the Error Correcting Code logic. This will also ensure that no corrupted sectors within SRAM are left thus setting clean space for data.
- Initialization of Stack Pointer and small data area (.sbss and sdata2) pointers: Since the PowerPC architecture doesn’t have a push/pop instruction for implementing a stack, the Embedded Application Binary Interface (EABI) conventions of stack frame creation and usage are followed. This allows an implementation in C-language that uses the stack to have support for parameter passing during function invocation, nonvolatile register preservation, local variables and code debugging.

  In this work, the stack pointer is initialized to the value of the address assigned to the memory region reserved for stack (available from linker script file). Such stack (defined at the top of internal SRAM) is just a designation of space for an actual stack frame that is created from memory set aside for use at run-time. General Purpose Register (GPR) R1 is dedicated as the stack frame pointer (SP).

  To take advantage of the PowerPC base plus displacement addressing mode, Small Data Areas (SDA) are used. The displacement is a signed 16-bit value, therefore 64 Kbytes may be addressed without changing the value in a base register. SDAs are useful for global and static variables and constants. GPR R2 is dedicated for use as a base pointer (anchor) for the read-only small data area (Containment of variables -> initialized: .sdata2, non-initialized: .sbss2). GPR R13 is dedicated for use as an anchor for addressing the read-write small data area. [23]

The bootloader initializes then these small data area anchor registers to the value of the addresses to which all data in the .sdata and .sbss sections can be addressed using a 16-bit signed offset. These values are available as macros automatically during linking.

- Initialization of interrupt vector and definition of branch table: To provide the MCU a mechanism for exception handling in form of traps, configure the interrupt vector offset registers (IVOR) for all the accepted interrupt sources, define Interrupt Service Routines (ISR) mapped to IVORs as well as enabling in the processor the recognition of interrupt events.
- Initialization and configuration of Frequency Modulated Phase Locked Loop (FMPLL): To be able to generate high system clocks from a crystal oscillator (16 MHz in this project). The configuration of the controlling parameters was set in order to get a system frequency up to 128 MHz, a proper value for flash reprogramming (between 25 MHz and a maximum operating frequency of 132 MHz with the crystal used).
- Initialization of Enhanced Modular Input/Output Subsystem: While completely optional, this module is initialized to provide an easy method for starting, reading and stopping time counts derived from structures known as channels.
- Initialization of External Bus Interface: To prepare the interface that controls data transfer and signals to the external SRAM module.
- Initialization of External SRAM: To prepare this memory space to store data.
- Copy from flash to RAM of initialized variables found at end of .text memory section: To have the data from .data and .sdata sections (corresponding to global and static variables) stored in its runtime address in RAM, since it's both readable and writable.
- Initialization of uninitialized data (clearing of .bss and .sbss sections): To ensure that these data doesn't contain corrupted values that would compromise the correct execution of code and therefore get unexpected results.
- Jump to "preparation for reprogramming" code: To start the process that sets the MCU into a preparation state to begin a reprogramming session.

3.1.2 Reprogramming of MCU (Secondary Bootloader)

Just as there is a part of code intended for the functions in a primary bootloader as recently explained, there's code that on its turn would deal with an eventual reception of a new application and therefore manage the erase and write operations performed on flash memory to store that application.

In concrete, for the present work the piece of software that handles the downloading of the target application and reprogramming of the flash memory is called the Secondary Bootloader (SBL). The secondary bootloader has to be downloaded to the target before any application data can be downloaded. It contains all necessary UDS functions and all flash routines that are needed to store the downloaded data onto the flash memory.

The primary bootloader will start and give all control over the processor to the secondary bootloader after it has been downloaded into the RAM memory. At first, the secondary bootloader initializes the Flash Bus Interface Unit (FBIU) on the target to enable read/write capabilities for the upcoming reprogramming operations. Right after that, it reinitializes the communication via CAN to be ready to receive UDS requests. When the flash reprogramming of the target is complete and it resets then the secondary bootloader will be automatically removed from RAM memory and the primary bootloader will again be in control after the reset cycle is complete.

3.1.3 Creating the bootloaders

To create the primary and the secondary bootloader two executable files are required as the output from the same project build. In order for the secondary bootloader to function properly it needs to be a part of the same memory layout as the primary bootloader, in the sense that it must "know" the start address of the reserved section to which the secondary bootloader shall be mapped. To achieve this, both bootloaders are programmed under the same project and the makefile used is programmed to instruct the compiler to compile the source code files for both the primary and secondary bootloaders on the same run and to be linked with the same linker script file. However, the set of object files for the primary bootloader is built separately into an executable file from that for the secondary bootloader. For convenience, the file that corresponds to the primary bootloader created is in *.elf format and is loaded onto the processor with a special programming tool. On its turn, the executable file that is created for the secondary bootloader is obtained in the Motorola .s19 format. This file includes sections defined for the primary bootloader in the range of addresses defined for it, but only the part containing the secondary bootloader is desired so the part corresponding to the primary bootloader file has to be cropped using a tool called SRecord and converted to binary format. This binary format is one of the three accepted file-formats (raw binary, intel-hex and motorola's .s19) for downloading boot software, application software or application data, as specified in [24]

3.2 Use of UDS

Based on the information presented in the theoretical background a functional and compatibility comparison was made to select between UDS and XCP as the method of choice for supervising the reprogramming process. The solid support provided by the existing specifications and especially by the AUTOSAR standard as of lately makes UDS an attractive alternative to work with. A detailed explanation of the result of the analysis can be seen in the section "Findings and Analysis".

3.3 UDS Messages

Every message includes at least one or two Protocol Control Information (PCI) byte(s) and one Service Identifier (SID) byte. PCI tells what type of frame it is and how many bytes of data it is in the frame. There are three types of frames.

- Single Frame
- First Frame
- Consecutive Frame

3.3.1 Frames

Single Frame (SF) is used when the whole message is shorter or equal to 7 bytes, PCI byte not included. If the message is longer than 7 bytes, then the message needs to be divided and send with multiple frames.

When sending multiple frames a First Frame (FF) is send first and then it is followed by Consecutive Frames (CF). The FF holds information about the length of the message (FF_DL) and the first six (6) bytes of data. The rest of the message is sent in CF and every CF consists of one sequence number (SN) and up to seven (7) bytes of data. The sequence number is used by the receiver to reassemble the message in the correct order after the reception is complete. [25]

	Byte 0		Byte 1	Byte 2-7
	High Nibble	Low Nibble		
SF	PCI TYPE	SF_DL	DATA	DATA
FF	PCI TYPE	FF_DL	FF_DL	DATA
CF	PCI TYPE	SN	DATA	DATA

Table 6 - Frame layout

UDS specifies that a CAN data frame shall always contain eight (8) bytes so if a request message doesn't fill eight bytes the remaining bytes are filled with 0x55. Response messages are filled with 0xAA instead of 0x55.

3.3.2 Implemented UDS services

- Diagnostic Session Control
- Communication Control
- Control DTC Settings
- Security Access
- Request Download
- Transfer Data
- Requests Transfer Exit
- Routine Control
- ECU Reset

3.3.2.1 Diagnostic Session Control

The ECU is executing in different modes (sessions) each with a specific set of diagnostic services enabled. This is to prohibit unauthorized persons to view ECU information or make changes to the ECU. There are two main types of sessions, default session and non-default session. The default session is, as the name says, the default state that the ECU is executing in. This session have only a limited number of diagnostic services enabled, but at least Diagnostic Session Control and ECU Reset. The Diagnostic Session Control service is used to switch between different sessions. Three session types have been implemented, default session, extended diagnostic session and programming session.
A server shall always start the default diagnostic session when powered up. If no other diagnostic session is started, then the default diagnostic session shall be running as long as the server is powered. One and only one session shall always be active in the server. [26]

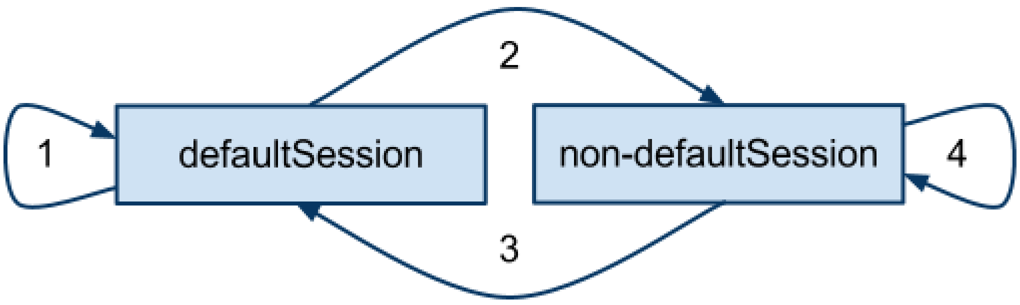


Figure 18 - Diagnostic Session transitions [26]

1. If the server is in defaultSession and the client requests to start defaultSession, then the server shall re-initialize defaultSession completely.
2. If the server is in defaultSession and the client requests to start non-defaultSession (extendedDiagnosticSession or programmingSession), then the server shall only reset the events that have been configured in the server via the ResponseOnEvent services before transition to an other session.
3. When the server transitions from non-defaultSession to the defaultSession, then the server shall reset each event that has been configured in the server via the ResponseOnEvent service and security shall be enabled. Any configured periodic scheduler shall be disabled and CommunicationControl and ControlDTCSetting services shall be reset to default state. The server shall reset all activated/initiated/changed settings/controls during the activated session.
4. If the server is in non-defaultSession and the client requests to start the same or another non-defaultSession, then the server shall (re-) initialize the non-defaultSession. Reset of events that has been configured in the server via the ResponseOnEvent service and enable Security shall be preformed.

Requests

- **Default Session (01 hex)**
  The default session is the normal state for the ECU to be in. Diagnostic services are not supported in this session only Diagnostic Session Control and ECU Reset. The ECU is always locked when it is in the default session.
- **Extended Session (02 hex)**
  All diagnostic services can be active in the extended session, it is implementation specific which services that are supported. Pre-programming services like Communication Control and Control DTC Setting are accessible here.
- **Programming Session (03 hex)**
  This session enables all services that are required to support reprogramming of an ECU e.g. Request Download, Routine Control and Transfer Data.

Data byte	**Parameter name**	**Hex value**
0	PCI	0x01
1	Diagnostic Session Request Service ID	0x10
2	Diagnostic Session Type	0x01-0x03

Table 7 - Diagnostic Session Control Request message layout

Example: Request Programming Session

Byte #	0	1	2	3	4	5	6	7
Value	0x01	0x10	0x02	0x55	0x55	0x55	0x55	0x55

Responses
The response message has a response SID and if it's a positive response the parameter is an echo of the request message parameter. If it is a negative response the parameter is one of three negative response codes.

Negative response codes:

- **subFunctionNotSupported** (12 hex)

  Function (session type) in the request message is not supported.

- **incorrectMessageLengthOrInvalidFormat** (13 hex)

  The length of the message is wrong.

- **conditionsNotCorrect** (22 hex)

  Used when the server is in a critical normal mode activity and therefore cannot perform the requested control functionality

Data byte	**Parameter name**	**Hex value**
0	PCI	0x01
1	Diagnostic Session Response Service ID	0x50
2	Diagnostic Session Type	0x01-0x03

Table 8 - Diagnostic Session Control Response message layout

Example: Positive response for programming session request

Byte #	0	1	2	3	4	5	6	7
Value	0x01	0x50	0x02	0xAA	0xAA	0xAA	0xAA	0xAA

3.3.2.2 Control DTC Setting

Shall be used to stop or resume the setting of diagnostic trouble codes in the ECU. [27]

Requests

- **On** (01 hex)
  The server shall resume the setting of diagnostic trouble codes according to normal operating conditions.
- **Off** (02 hex)
  The server shall stop the setting of diagnostic trouble codes.

Data byte	Parameter name	Hex value
0	PCI	0x01
1	DTC Setting Request Service ID	0x85
2	DTC Setting Type	0x01-0x02

Table 9 - Control DTC Settings Request message layout

Example: Request Control DTC Settings off

Byte #	0	1	2	3	4	5	6	7
Value	0x01	0x85	0x02	0x55	0x55	0x55	0x55	0x55

Responses
The response message has a response SID and if it's a positive response the parameter is an echo of the request message DTCsettingType parameter. If it is a negative response the parameter is one of four negative response codes.

Negative response codes:

- **subFunctionNotSupported** (12 hex)
  Function (session type) in the request message is not supported.
- **incorrectMessageLengthOrInvalidFormat** (13 hex)
  The length of the message is wrong.
- **conditionsNotCorrect** (22 hex)
  Used when the server is in a critical normal mode activity and therefore cannot perform the requested DTC control functionality
- **requestOutOfRange** (31 hex)
  The server shall use this response code if it detects an error in the DTCSettingControlOptionRecord.

Data byte	**Parameter name**	**Hex value**
0	PCI	0x01
1	DTC Settings Response Service ID	0xC5
2	DTC Settings Type	0x01-0x02

Table 10 - Control DTC Settings Response message layout

Example: Positive response for Control DTC Settings request

Byte #	0	1	2	3	4	5	6	7
Value	0x01	0xC5	0x02	0xAA	0xAA	0xAA	0xAA	0xAA

3.3.2.3 Communication Control

This service is used to switch on/off reception and/or transmission of certain messages. [28]

Requests

The Communication Type parameter is used to reference the kind of communication to be controlled. The parameter is a bit-code value which allows control of multiple communication types at the same time.

- **Normal Communication Messages** (0x01)
- **Network Management Communication Messages** (0x02)
- **Network Management Communication Messages and Normal Communication Messages** (0x03)

Control Type parameters:

- **enableRxAndTx** (00 hex)
  This value indicates that the reception and transmission of messages shall be enabled for the specified communication type.
- **enableRxAndDisableTx** (01 hex)
  This value indicates that the reception of messages shall be enabled and transmission shall be disabled for the specified communication type.
- **disableRxAndEnableTx** (02 hex)
  This value indicates that the reception of messages shall be disabled and transmission shall be enabled for the specified communication type.
- **disableRxAndTx** (03 hex)
  This value indicates that the reception and transmission of messages shall be disabled for the specified communication type.

Data byte	**Parameter name**	**Hex value**
0	PCI	0x01
1	Comm Control Request Service ID	0x28
2	Comm Type	0x01-0x03
3	Control Type	0x00-0x03

Table 11 - Communication Control Request message layout

Example: Request Communication Control enables Rx and Tx for normal communication messages

Byte #	0	1	2	3	4	5	6	7
Value	0x01	0x28	0x01	0x00	0x55	0x55	0x55	0x55

Responses
The response message has a response SID and if it's a positive response the parameter is an echo of the request message control type parameter. If it is a negative response the parameter is one of four negative response codes.

Negative response codes:

- **subFunctionNotSupported** (12 hex)

  Function (session type) in the request message is not supported.

- **incorrectMessageLengthOrInvalidFormat** (13 hex)

  The length of the message is wrong.

- **conditionsNotCorrect** (22 hex)

  Used when the server is in a critical normal mode activity and therefore cannot disable/enable the requested communication type

- **requestOutOfRange** (31 hex)

  The server shall use this response code if it detects an error in the communicationType parameter.

Data byte	**Parameter name**	**Hex value**
0	PCI	0x01
1	Comm Control Response Service ID	0x68
2	Control Type	0x00-0x03

Table 12 - Communication Control Response message layout

Example: Positive response for Communication Control request

Byte #	0	1	2	3	4	5	6	7
Value	0x01	0x68	0x00	0xAA	0xAA	0xAA	0xAA	0xAA

3.3.2.4 Security Access

To prevent that the ECU is modified by unauthorized persons most UDS services are locked. To get access to services that are used to modify the ECU the user first has to grant access through the Security Access Service. Only after the security access service have been passed, services like Request Download and Transfer Data can be used. The security concept used is called "Seed and Key". [29]
Security Access Service flow:

1. The client sends a request for a "seed" to the server that it wants to unlock.
2. The server replies by sending the "seed" back to the client.
3. The client then generates a "key" based on the "seed" and sends the key to the server.
4. If the client generated the "key" with the correct algorithm the server will respond that the "key" was valid and that it will unlock itself.

Data byte	Parameter name	Hex value
0	PCI	0x01
1	Security Access SID	0x27
2	Request Seed/Send key	0x01/0x00-0xFF
.. n	Send key	0x00-0xFF

Table 13 - Security Access Request message layout

Example: Request Seed

Byte #	0	1	2	3	4	5	6	7
Value	0x01	0x27	0x01	0x55	0x55	0x55	0x55	0x55

Response
The response message has a response SID and if it is a positive response the parameter is an echo of the request message parameter. If it is a negative response the parameter is one of eight negative response codes.

Negative response codes:

- **subFunctionNotSupported** (12 hex)

  Function (session type) in the request message is not supported.

- **incorrectMessageLengthOrInvalidFormat** (13 hex)

  The length of the message is wrong.

- **conditionsNotCorrect** (22 hex)

  Used when the server is in a critical normal mode activity and therefore cannot perform the requested DTC control functionality

- **requestSequenceError** (24 hex)

  Send if the "sendKey" sub-function is received without first receiving a "requestSeed" request message.

- **requestOutOfRange** (31 hex)

  The server shall use this response code if it detects an error in the DTCSettingControlOptionRecord.

- **invalidKey** (35 hex)

  Send if an expected "sendKey" sub-function value is received and the value of the key does not match the server's internally stored/calculated key.

- **exceededNumberOfAttempts** (36 hex)

  Send if the delay timer is active due to exceeding the maximum number of allowed false access attempts.

- **requiredTimeDelayNotExpired** (37 hex)

  Send if the delay timer is active and a request is transmitted.

Data byte	Parametern name	Hex value
0	PCI	0x01
1	Security Access RSID	0x67
2	Request Seed	0x01

Table 14 - Security Access Response message layout

Example: Positive response for security access request

Byte #	0	1	2	3	4	5	6	7
Value	0x01	0x67	0x02	0x01	0x02	0xAA	0xAA	0xAA

3.3.2.5 Request Download

The request download service is called when data is to be transferred to the ECU. The request shall contain the size of the data to be transferred and the address to where it shall be placed. The ECU responds with the size of its buffer so that the sender can divide the data into appropriate sized blocks and send them one at a time. [30]

Requests

There are four different request parameters that shall be included in the request.

dataFormatIdentifier

This data parameter is a one-byte value with each nibble encoded separately. The high nibble specifies the "compressionMethod" and the low nibble specifies the "encryptingMethod". The value 0x00 specifies that no compressionMethod or encryptingMethod is used.

addressAndLengthFormatIdentifier

bit 7 - 4: Length (number of bytes) of the memorySize parameter.

bit 3 - 0: Lenght (number of bytes) of the memoryAddress parameter.

memoryAddress

The parameter memoryAddress is the starting address of the server memory to which the data is to be written.

memorySize (unCompressedMemorySize)

This parameter shall be used by the server to compare the uncompressed memory size with the total amount of data transferred during the TransferData service.

Data byte	**Parametern name**	**Hex value**
0	PCI	0x01
1	Request Download SID	0x34
2	dataFormatIdentifier	0x00-0xFF
3	addressAndLengthFormatIdentifier	0x00-0xFF
4..n	memoryAddress	0x00-0xFF
n..m	memorySize	0x00-0xFF

Table 15 - Request Download Request message layout

Responses

Positive response codes:

- **lengthFormatIdentifier**

  bit 7 - 4: length (number of bytes) of the maxNumberOfBlockLength parameter.

  bit 3 - 0: reserved by document, to be set to 0 hex.

- **maxNumberOfBlockLength**

  This parameter is used by the requestDownload positive response message to inform the client how many data bytes (maxNumberOfBlockLength) shall be included in each TransferData request message from the client.

This length reflects the complete message length, including the service identifier and the data parameters present in the TransferData request message. This parameter allows the client to adapt to the receive buffer size of the server before it starts transferring data to the server.

Data byte	Parametern name	Hex value
0	PCI	0x01
1	Reques Download RSID	0x74
2	lengthFormatIdentifier	0x00-0xFF
3	maxNumberOfBlockLength	0x00-0xFF

Table 16 - Request Download Response message layout

Negative response codes:

- **incorrectMessageLengthOrInvalidFormat** (13 hex)
  The length of the message is wrong.
- **conditionsNotCorrect** (22 hex)
  This return code shall be sent if a server receives a request for this service while in
  the process of receiving a download of a software or calibration module. This could
  occur if there is a data size mismatch between the server and the client during the
  download of a module.
- **requestOutOfRange** (31 hex)
  This return code shall be sent if
  1) the specified dataFormatIdentifier is not valid,
  2) the specified addressAndLengthFormatIdentifier is not valid, or
  3) the specified memoryAddress/memorySize is not valid.
- **securityAccessDenied** (33 hex)
  This return code shall be sent if the server is secure (for servers that support the
  SecurityAccess service) when a request for this service has been received.

- **uploadDownloadNotAccepted** (70 hex)
  This response code indicates that an attempt to download to a server's memory
  cannot be accomplished due to fault conditions.

3.3.2.6 *Transfer Data*

The Transfer Data service receives blocks of data and check so the blocks are received in the right order. If the right block is received, then it is written to correct memory location and a positive response is send. [31]

Requests

blockSequenceCounter

The blockSequenceCounter parameter value starts at 01 hex with the first TransferData request that follows the RequestDownload (34 hex) service. Its value is incremented by 1 for each subsequent TransferData request. At the value of FF hex, the blockSequenceCounter rolls over and starts at 00 hex with the next TransferData request message.

Data byte	Parametern name	Hex value
0	PCI	0x01/0x10/0x20-0x2F
1	Transfer Data SID	0x36
2	blockSequenceCounter	0x00-0xFF
4..n	data	0x00-0xFF

Table 17 - Transfer Data Request message layout

Responses

Positive response codes:

- **blockSequenceCounter**
- This parameter is an echo of the blockSequenceCounter parameter from the request message.

Data byte	Parametern name	Hex value
0	PCI	0x01/0x10/0x20-0x2F
1	Transfer Data SID	0x76
2	blockSequenceCounter	0x00-0xFF

Table 18 - Transfer Data Response message layout

Negative response codes:

- **incorrectMessageLengthOrInvalidFormat** (13 hex)

  The length of the message is wrong (e.g. message length does not meet the requirements of the maxNumberOfBlockLength parameter returned in the positive response to requestDownload).

- **requestSequenceError** (24 hex)

  The server shall use this response code:

  - If the RequestDownload service is not active when a request for this service is received.
  - If the RequestDownload service is active, but the server has already received all data as determined by the memorySize parameter in the active RequestDownload or RequestUpload service.

- **requestOutOfRange** (31 hex)

  This return code shall be sent if the transferRequestParameterRecord contains additional control parameters (e.g. additional address information) and this control information is invalid

- **transferDataSuspended** (71 hex)

  This return code shall be sent if:

  - The response code indicates that a data transfer operation was halted due to a fault.
  - The download module length does not meet the requirements of the memorySize parameter sent in the request message of the requestDownload service.

- **generalProgrammingFailure** (72 hex)

  This return code shall be sent if the server detects an error when erasing or programming a memory location in the permanent memory device (e.g. Flash Memory) during the download of data.

- **wrongBlockSequenceCounter** (73 hex)

  This return code shall be sent if the server detects an error in the sequence of the blockSequenceCounter.

The repetition of a TransferData request message with a blockSequenceCounter equal to the one included in the previous TransferData request message shall be accepted by the server.

3.3.2.7 Request Transfer Exit

When the transfer of data is complete a message to the Request Transfer Exit service is send. If Transfer Data is complete and have received all data a positive response is send back. [32]

Request

Data byte	Parametern name	Hex value
0	PCI	0x01
1	Request Transfer Exit SID	0x37

Table 19 - Request Transfer Exit Request message layout

Response

Data byte	Parametern name	Hex value
0	PCI	0x01
1	Request Transfer Exit RSID	0x77

Table 20 - Request Transfer Exit Response message layout

Negative response codes:

- **incorrectMessageLengthOrInvalidFormat** (13 hex)

  The length of the message is wrong

- **requestSequenceError** (24 hex)

  The server shall use this response code:

  - The programming process is not completed when a request for this service is received
  - The RequestDownload service is not active

3.3.2.8 Routine Control

The Routine Control service is used to start or stop a routine or to request routine results. [33]

Requests

routineControlType

- **startRoutine** (01 hex)
- **stopRoutine** (02 hex)
- **reqestRoutineControl** (03 hex)

routineIdentifier

This parameter identifies a server local routine.

Data byte	Parametern name	Hex value
0	PCI	0x01
1	Routine Control SID	0x31
2	routineIdentifier	0x00-0xFF
3	routineControlType	0x01-0x03

Table 21 - Routine Control Request message layout

Responses

Positive

- **routineControlType**

  This parameter is an echo of bits 6 - 0 of the sub-function parameter from the request message

- **routineIdentifier**

  This parameter is an echo of the routineIdentifier from the request message

Negative response codes:

- **subFunctionNotSupported** (12 hex)
  This code is returned if the requested sub-function is not supported.
- **incorrectMessageLengthOrInvalidFormat** (13 hex)
  The length of the message is wrong.
- **conditionsNotCorrect** (22 hex)
  This code shall be returned if the criteria for the request RoutineControl are not met.
- **requestSequenceError** (24 hex)
  This code shall be returned if the “stopRoutine” subfunction is received without first receiving a “startRoutine” for the requested routineIdentifier.
- **requestOutOfRange** (31 hex)
  This code shall be returned if the server does not support the requested routineIdentifier.
- **securityAccessDenied** (33 hex)
  This code shall be sent if this code is returned if a client sends a request with a valid secure routineIdentifier and the server’s security feature is currently active.
- **generalProgrammingFailure** (72 hex)
  This return code shall be sent if the server detects an error when performing a routine, which accesses server internal memory. An example is when the routine erases or programmes a certain memory location in the permanent memory device (e.g. Flash Memory) and the access to that memory location fails.

3.3.2.9 ECU Reset

To do a hard reset of the ECU, the ECU Reset service can be used by sending a ECUReset request message. [34]

Data byte	Parametern name	Hex value
0	PCI	0x01
1	ECU Reset SID	0x11
2	resetType	0x01-0x05

Table 22 - ECU Reset Request message layout

Requests

Reset Type

- **hardReset** (01 hex)
  Hard reset message triggers a reset that simulates start-up sequence that are executed when the processor is power up after the power has been disconnected.
- **keyOffOnReset** (02 hex)
  Shall be used when a condition similar to that the driver turns off and on the ignition key. The reset shall simulate an interruption on the switched power supply.
- **softReset** (03 hex)
  This reset restarts the application program.
- **enableRapidPowerShutDown** (04 hex)
  Enables so that the ECU transition to sleep mode after the ignition is turned off.
- **disableRapidPowerShutDown** (05 hex)
  Disables "transition to sleep mode after ignition is turned off".

Responses

The positive response code for ECU Reset consists of the requested reset type and the parameter *powerDownTime* that indicates how long time in seconds the ECU will remain in the power down sequence before entering sleep mode.

Data byte	**Parametern name**	**Hex value**
0	PCI	0x01
1	ECU Reset RSID	0x51
2	resetType	0x01-0x05
3	Power Down Time	0x00-0xFF

Table 23 - ECU Reset Response message layout

Negative response codes:

- **subFunctionNotSupported** (12 hex)
  This code is returned if the requested sub-function is not supported.
- **incorrectMessageLengthOrInvalidFormat** (13 hex)
  The length of the message is wrong.
- **conditionsNotCorrect** (22 hex)
  This code shall be returned if the criteria for the request RoutineControl are not met.
- **securityAccessDenied** (33 hex)
  This code shall be sent if this code is returned if a client sends a request with a valid secure routineIdentifier and the server's security feature is currently active.

Example: Positive response for ECU Reset request

Byte #	0	1	2	3	4	5	6	7
Value	0x01	0x51	0x01	0x00	0xAA	0xAA	0xAA	0xAA

3.4 Tools Used

In this section is provided a brief description of the tools employed during the development and test of the bootloaders.

3.4.1 Arctic Studio

The Integrated Development Environment (IDE) used to write and debug code was Arctic Studio in its simple version, from the company ArcCore; a software-developing company whose product line goes from developing tools, code builders and verifiers, to prototype boards. Their products have a strong focus on AUTOSAR compliance.

Arctic Studio is built on a well-known IDE; the industry standard Eclipse. Because of this, besides the common capabilities that it offers for writing and editing source code, it counts for example with GCC cross compilers for supported platforms. It also features source code version management plugins such as SVN or GIT.

The full version of Arctic Studio includes some support tools that cover in great extent the development of software for a complete AUTOSAR based implementation. Among these tools, the following are the most relevant:

- ***BSW Builder*:** Basic Software Builder for editing and generating BSW configurations for AUTOSAR platforms. It helps on the process of configuration of Arctic Core components, validating the components and generating configuration files.
- ***RTE Builder:*** Real Time Environment Builder that provides configuration of RTE, mapping of runnable tasks, validation to ensure that RTE actually runs and generation of RTE source and header files.
- ***Extract Builder:*** In this tool, the ECU Integrator feature defines the software architecture with all the software components, ports and connections between them, thus helping to build an AUTOSAR ECU software model.
- ***SWC:*** The Software Component Builder aids on creating, importing, modifying and validating software components, port interfaces and data types.

For the present work, Arctic Studio was configured to use a GCC (GNU Cross Compiler) from the PowerPC EABI toolchain.

3.4.2 PEEDI

PEEDI stands for Powerful Embedded Ethernet Debug Interface and it is a very useful tool for debugging and downloading program code to the target processor via the JTAG port. JTAG is a protocol standardized by the IEEE that allows having full control over the CPU core. Figure [25] shows the appearance of PEEDI, as developed and commercialized by Ronetix, an austrian company.

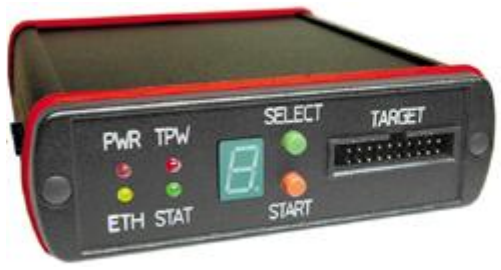


Figure 19: View of PEEDI device

In this project, PEEDI was used to start and stop processor's execution, examine and store values in registers, to run special commands to manipulate the behavior and to store program code in the target's flash memory, MPC5567 in this case. Most of the commands are entered in a Command Line Interface (CLI) that instructs PEEDI what to do. More information about PEEDI's complete set of features and configuration can be found at: *www.ronetix.at/PEEDI.html*

By using the developer's PC as a host computer with an installed program that works as a file server (via TFTP for example) PEEDI will be allowed to control and retrieve configuration files or executable images. The layout of the connection used in this work resembles the one shown in Figure 26

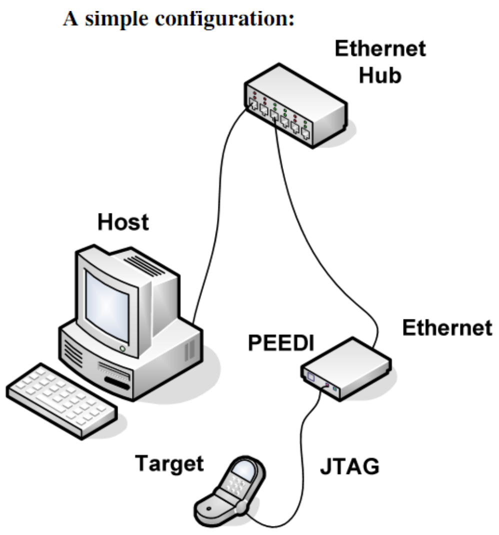


Figure 20: Layout of connections for communicating with PEEDI and target

The file server program used for this development is called TFTP32. To start communicating with the PEEDI, the TFTP32 program must be first set up and running with the working directory and right IP address of the host. Then a telnet session is run with the IP address given to the PEEDI as an argument. Once the session is running when connected, it is possible to start executing commands directly on the processor.

Worth to mention is the fact that for operating on the target processor, PEEDI needs to be loaded a configuration file that describes the specifics of the target: CPU type, flash type and metrics, RAM address and size, etc. In this case, PEEDI was previously loaded this configuration file with the specific information for MPC5567.

For debugging from Arctic Studio via PEEDI, a preliminary configuration is also required on the IDE and this is done by first selecting the command for hardware debugging available from the GCC toolchain (GDB) which PEEDI recognizes, then choosing a generic TCP/IP method to reach the JTAG device (PEEDI in this case) and providing host computer's IP address and port.

3.4.3 SRecord

As described by its creator, Peter Miller, SRecord is a program intended for manipulating EPROM loadable files. Some of the file formats supported by the latest version (1.59) of SRecord are: ascii-hex, binary, asm, C-array, basic, hexdump, intel, Motorola s-record, vhdl, etc. Included in the SRrecord package there are some tools that offer different capabilities:

- srec_cat: Used to concatenate (join) load files or portions of them together. It's also used to convert files from one format to another. Some filtering options are also available within it.
- srec_cmp: Used to compare load files or portions of them to analyze equality.
- srec_info: Used to print summary information about load files.

From these, srec_cat was the tool employed for resizing the built executable *.s19 file corresponding to the code for the secondary bootloader by using the filter option "crop" and for further converting the resized output into a *.bin binary file.

Srec_cat runs directly on DOS, it's easy to use and it proved to be a very handy and illustrating tool for manipulating loadable files. More detailed information can be found at: *http://srecord.sourceforge.net/*

3.4.4 CANalyzer and CANcaseXL

CANalyzer is a development tool from Vector GmbH for CAN bus based systems that allows the user to observe, analyze and provide data traffic on the bus line. Specifically the functions included range from listing bus data traffic (tracing), displaying data segments of specific messages, transmitting predefined messages and replaying recorded messages, to statistically evaluating messages, bus loading and disturbances. The picture below illustrates the main window after opening CANalyzer.

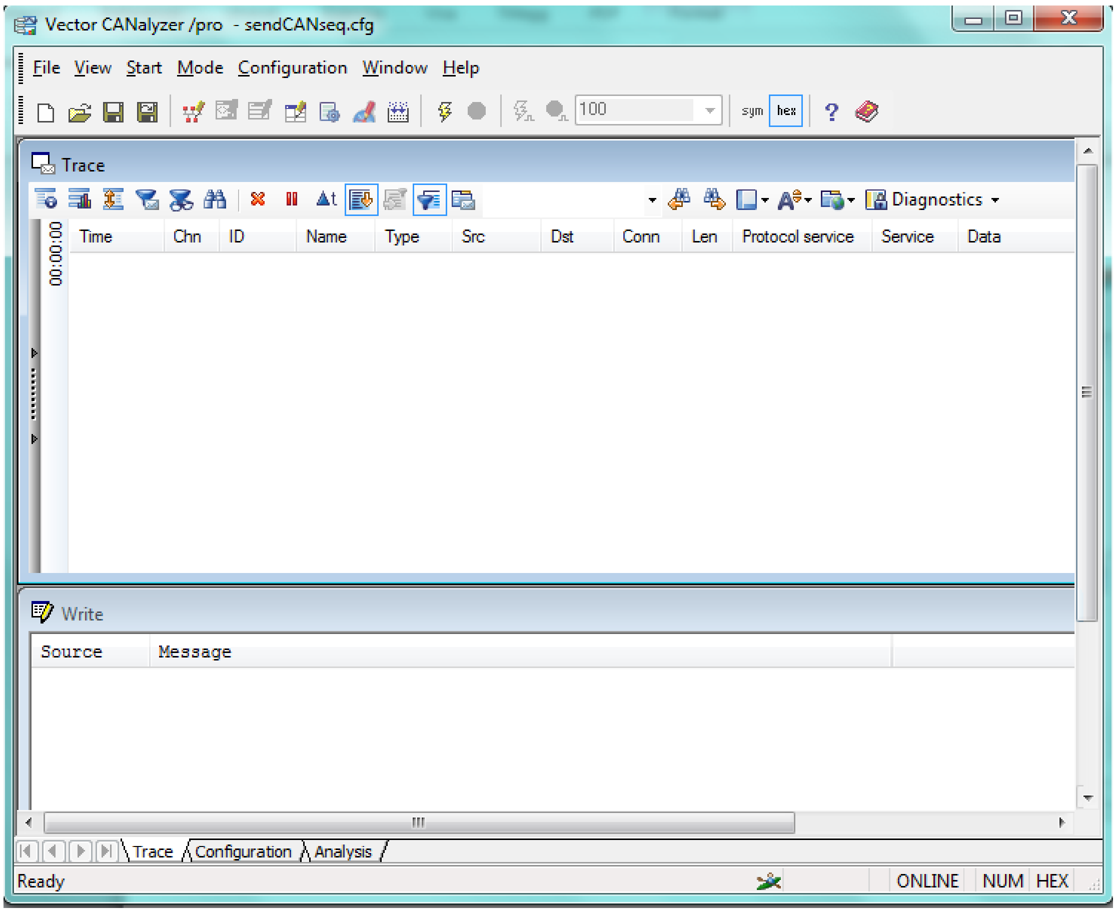


Figure 21: Main window in CANalyzer ver.7.5.66

To start using CANalyzer the bus was first setup. In this case the hardware interface CANcaseXL USB 2.0 was provided for using it as the gateway that, when connected to the bus, made possible the transmission and reception of data frames to and from the target platform QR5567.

The main features of CANcaseXL include the USB 2.0 interface for connecting to a laptop or PC, a 32-bit microcontroller with 64 MHz, 2 communication channels, CAN 2.0B and LIN controllers and the bus transceivers which compose the hardware that handles the signals in the bus at the physical level. The associated driver for CANcaseXL was also installed in the computer running CANalyzer.

This device can be fully operated from CANalyzer. Figure 22 shows CANcaseXL and the associated cable to connect between it and the work computer.

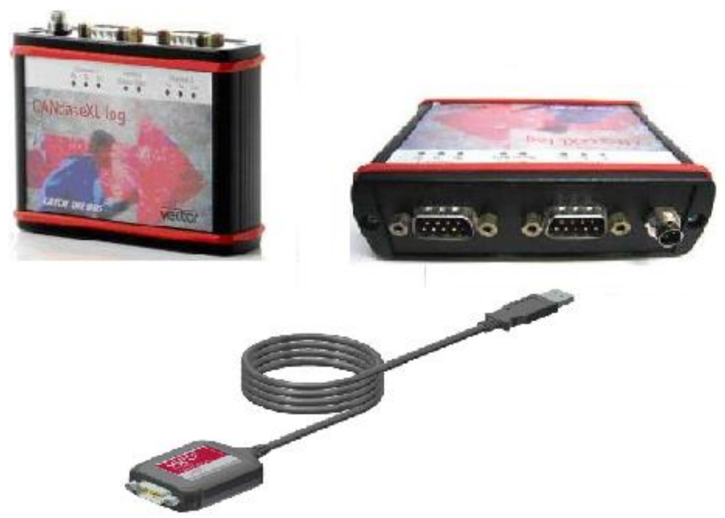
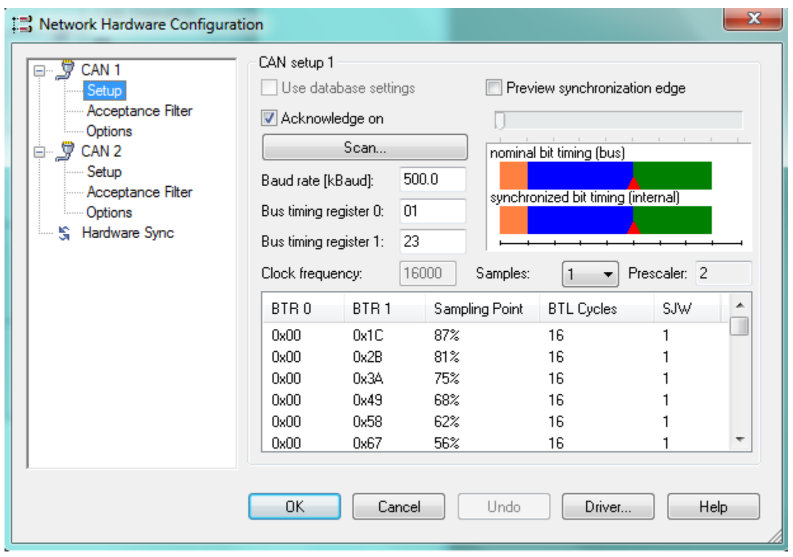


Figure 22: CANcaseXL device and interface cable

To finish the bus physical setup, a D-sub-9 connector was wired to the CAN-High, CAN-Low and Ground pins of the CAN ports on the QR5567 platform to be able to connect the interface cable to it, since such connector type was missing. Back into CANalyzer, the configuration of the bus properties was done by setting the communication Baud Rate to 500 kBaud (about half the maximum); other parameters like the Bus timing registers 0 and 1 and the amount of samples were left with default values. The picture in Figure 23 shows the appearance of this configuration window. Alternatively, pressing Scan gives current bus parameters.



Use database settings [ ]
Preview synchronization edge [ ]
Acknowledge on [✓]

Baud rate [kBaud]: 500.0
Bus timing register 0: 01
Bus timing register 1: 23
Clock frequency: 16000
Samples: 1
Prescaler: 2

BTR 0	BTR 1	Sampling Point	BTL Cycles	SJW
0x00	0x1C	87%	16	1
0x00	0x2B	81%	16	1
0x00	0x3A	75%	16	1
0x00	0x49	68%	16	1
0x00	0x58	62%	16	1
0x00	0x67	56%	16	1

Figure 23: Network Hardware Configuration Window

The next preparing step consisted of creating the data flow. This is achieved by opening the Measurement Setup menu. The diagram presented includes a data source, symbolized by the PC card to the left (corresponding to CANcaseXL in this case), and some evaluation blocks to the right. In this sense, the flow of data goes from left to right. Connection lines and branches are visible between both sides to ease the understanding of the possible courses of data flow. See Figure 24.

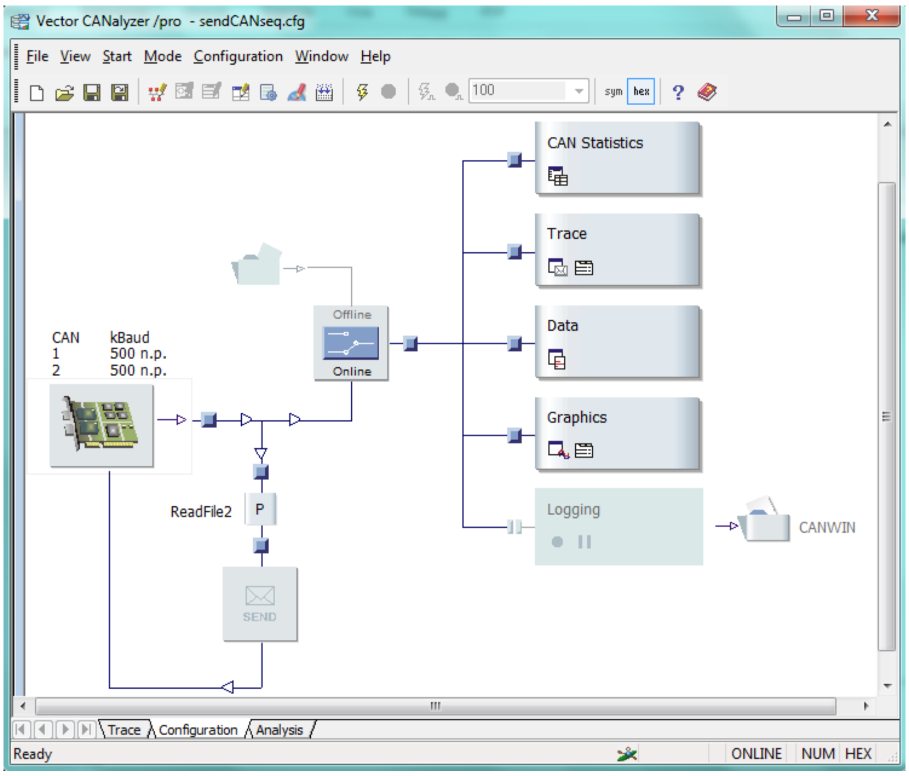
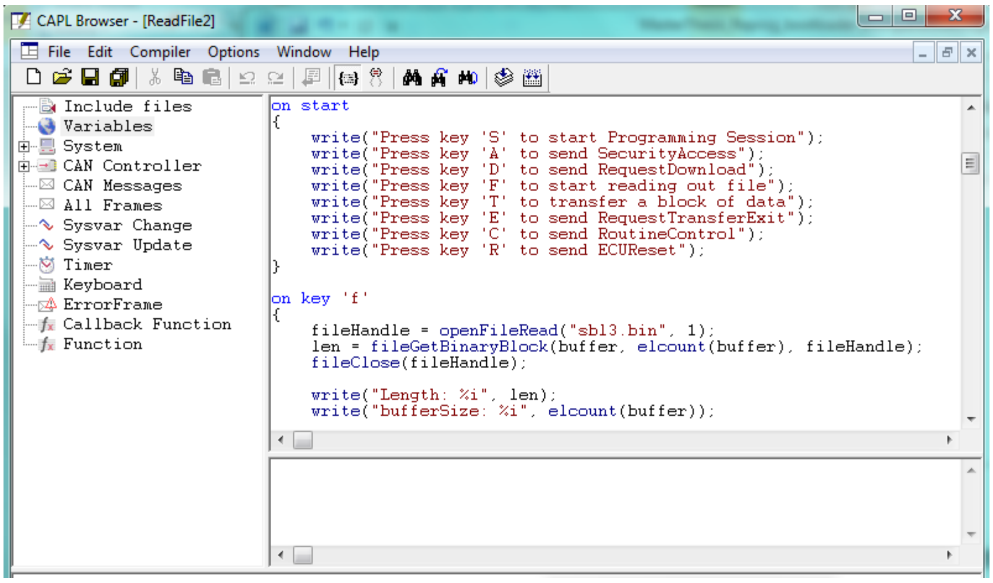


Figure 24: Measurement Setup Configuration Window

The small rectangles in the diagram are identifies as "hotspots" and they are used to insert additional blocks for manipulating the data flow. One type of these, the program blocks (identified with letter "P"), can be inserted at any point in the data flow diagram. They are edited with a C-like language named "CAPL" by using an IDE called CAPL Browser that is contained in CANalyzer. This project benefited of using the CAPL programming language (and its already available functions) in the development of code in charge of carrying the sequence of transmission of UDS requests and blocks of program data within the binary file, to the target. The image in Figure 25 shows the main window in the CAPL Browser.



```
on start
{
    write("Press key 'S' to start Programming Session");
    write("Press key 'A' to send SecurityAccess");
    write("Press key 'D' to send RequestDownload");
    write("Press key 'F' to start reading out file");
    write("Press key 'T' to transfer a block of data");
    write("Press key 'E' to send RequestTransferExit");
    write("Press key 'C' to send RoutineControl");
    write("Press key 'R' to send ECUReset");
}

on key 'f'
{
    fileHandle = openFileRead("sbl3.bin", 1);
    len = fileGetBinaryBlock(buffer, elcount(buffer), fileHandle);
    fileClose(fileHandle);

    write("Length: %i", len);
    write("bufferSize: %i", elcount(buffer));
```

Figure 25: CAPL Browser

4 Findings and Analysis

4.1 Comparison between UDS and XCP

UDS (ISO 14229-1) have been a part of the AUTOSAR specification since the beginning and many of its functions are now supported by the latest AUTOSAR specification 4.0. XCP was officially introduced in version 4.0 of the AUTOSAR specification and doesn’t have the same level of functionality implemented in AUTOSAR as UDS has.

As an outcome of the research analysis presented in the Theoretical Background and emphasized under Method and Implementation, the following comparison table is included to show which types of functions are supported by the UDS protocol and the XCP protocol. It also indicates if the function in turn is implemented in AUTOSAR.
Since the compliance with the AUTOSAR standard is one of the driving axes of the present work, the degree of coverage of the standard has a strong effect in the choice of one of the two alternatives. From the Table 24 below the following are considered the most decisive factors in the selection of UDS over XCP:

Flash Programming
In AUTOSAR version 4.0 some of the functions that are used for flash programming via UDS are implemented but the support is not complete. XCP on the other hand only had its basic functions (officially) implemented in AUTOSAR 4.0 and one of the paragraphs in the section 4.1 “Limitations” in the S*pecifications of module XCP* AUTOSAR-document states:
“*The XCP feature “Flash Programming for ECU development purposes” is currently out of AUTOSAR scope*”. [35] No AUTOSAR roadmaps indicate that remote flash programming is to be added to the AUTOSAR specification in the coming revisions or years.

Download of Data
The XCP module as specified in AUTOSAR does not include any of the most fundamental functions needed for flash programming unlike the UDS module which counts with services such as *RequestDownload* and*TransferData*. The *RequestDownload* service by itself is used by the client to initiate a data transfer from the client to the server. [10] Very important parameters like the starting address and size of the memory to be reprogrammed are included within this service. Such basic functionality is not met by XCP according to AUTOSAR.

Standard in OBD
UDS is a part of the international standard *ISO 15765: Road vehicles — Diagnostics on Controller Area Networks (CAN)* which further is one of the standards in the widely spread OBD-II standard for On-Board Diagnostics. This indicates that UDS already is widely accepted standard for diagnostic communication in the automotive industry. [4]

Function	Supported by UDS [9]	Supported by XCP [16]	Supported by UDS in AUTOSAR [10]	Supported by XCP in AUTOSAR [35]
Flash programming	✔	✔	✘	✘
Download of data	✔	✔	✔	✘
Upload of data	✔	✔	✔	✘
Reset function	✔	✔	✔	✔
Accept data transfer state	✔	✔	✔	✔
Access Security	✔	✔	✔	✔
Read parameters	✔	✔	✔	✔
Write parameters	✔	✔	✔	✔
Standard in OBD	✔	✘	✔	✘
CAN interface	✔	✔	✔	✔
LIN interface	✔	✘	✔	✘
FlexRay interface	✔	✔	✔	✔
Ethernet interface	✔	✔	✘	✔

Table 24 - Comparison of functions between UDS and XCP

4.2 Initialization and startup

A simple program that instructs the available LED on the QR5567 to infinitely blink with a defined frequency was written to verify that some sort of code could run without crashing after the initialization. A greater test was carried and passed when the rest of the code for the primary bootloader was executed and critical resources like timing, interrupts, or the ability to run directly on RAM were required to function well.

By working this way it was proved that a mixed approach towards the design and implementation of the bootloader payed off. This means setting up the most essential features in the microprocessor in addition to those that the "specialized" part of the primary bootloader code needed for preparing for reprogramming.

4.3 Communication

The implementation made possible to establish communication over the CAN bus between the target platform QR5567 and the client program CANalyzer via CANcaseXL. In addition, the diagnostic session for reprogramming can be started in the target and the sequence of associated UDS requests sent are served and responded by it. The following subsections describe firstly the used modules as defined by AUTOSAR which were implemented or participated in the communication process (on the server side) and later on the process for carrying the diagnostic session according to UDS.

4.3.1 Data flow through layers

UDS messages, and other data, have to pass through the different layers that AUTOSAR consists of before they can be handled by the correct UDS service.

$$\begin{tikzcd}
\text{UDS Services} \arrow[d, leftrightarrow] \\
\text{DCM} \arrow[d, leftrightarrow] \\
\text{PDU Router} \arrow[d, leftrightarrow] \\
\text{CAN TP} \arrow[d, leftrightarrow] \\
\text{CAN Interface} \arrow[d, leftrightarrow] \\
\text{CAN Driver} \arrow[d, leftrightarrow] \\
\text{Physical Bus}
\end{tikzcd}$$

4.3.1.1 The Physical Bus

The physical bus consists of the wiring and the hardware that connects the nodes in a CAN network. The data is represented by high and low voltage levels on the bus that are translated into 1 and 0 in the CAN transceiver. When the data is received it is placed in one of the 64 CAN buffers that are free and a receive interrupt is generated to notify to the upper layer that a messages can be fetched. When data have been placed in a transmit buffer by the CAN driver it is translated by the transceiver and sent out on the bus. When it is done it will generate a transmit interrupt to tell the layer above that the message have been sent away.

4.3.1.2 CAN Driver

When a CAN interrupt has been generated the CAN driver searches through the buffer and fetches the CAN message and sends it to the CAN Interface layer. If the CAN driver gets a message from CAN interface it will place the message in a transmit buffer and tell the transceiver to send it out on the bus. The CAN driver will also pass on the confirmation that the message has been transmitted successfully.

4.3.1.3 CAN Interface

Upon reception of a message from the CAN driver CAN interface will translate the CAN Id to a Protocol Data Unit Identifier (PDU Id). The CAN interface will then see if the PDU Id is a valid one and if it is send the message to the layer above, CAN Tp. When a message is to be transmitted, CAN interface will prepare the CAN message frame and add the CAN Id, length and data to the frame. The message frame is then sent to the CAN driver. The transmit confirmation message will be forwarded to the CAN Tp layer.

4.3.1.4 CAN Tp

When the CAN Transport layer receives a message from the interface layer it will read the Protocol Control Information (PCI) of the message to see if it is a single frame, first frame or a consecutive frame. Depending of what type of frame it is it will call different functions in the PDU Router layer.

4.3.1.5 PDU Router

The role of PDU Router is to route messages from all the communication interface modules up to the correct layer above. All diagnostic messages will be forwarded to the DCM module. The PDU Router does not modify or even care about the context of the frame, it will just look at the PDU Id to determine to what module it shall deliver the message. The same applies for when a message is being transmitted, the PDU Router will look at the PDU Id and forward the message the the correct communication interface.

4.3.1.6 DCM

When the messages arrive to the Diagnostic Communication Manager module they are examined to see what type of diagnostic service that shall be activated. The DCM also keeps track of what diagnostic session the ECU is in and if the requested diagnostic service is available in this session. If the service is not available then will the DCM module send a negative response message back to the client who sent the request If the request or the received data is transmitted in multiple consecutive frames, they are put together into one message here in the DCM module before the data is sent to the correct diagnostic service.

4.3.1.7 UDS Services

The diagnostic service handles the request and performs the action that has been requested. If the received request is faulty or if it cannot be handled by the service, a negative response message will be sent back with an appropriate response code.

4.3.2 Data flow through layers, Example:

This example describes the reception of data that are transferred in multiple frames, figure 19.

When the CAN Interface receives an interrupt that indicates that a new CAN message have been received it will start to send the data to the CanTp_RxIndication function. At the arrival of the first message to the CanTp_RxIndication the CAN Tp module will call the PduR_StartOfReception function. The PDU Router will just pass on the call to the Dcm_StartOfReception function. In Dcm_StartOfReception a buffer will be prepared that can hold the data that will arrive and a response will be sent back to tell that the buffer is ready.

When CAN Tp receives the next message it will send the message to the PduR_CopyRxData function. The PDU Router will again just pass on the call to the Dcm_CopyRxData function. In Dcm_CopyRxData the data will be extracted from the message and stored into the buffer. The CopyRxData sequence will continue until all data have been received and placed in the buffer. When all data have been received, CAN Tp will call PduR_RxIndication function which in turn will call Dcm_RxIndication function to tell that the copy of data is complete.

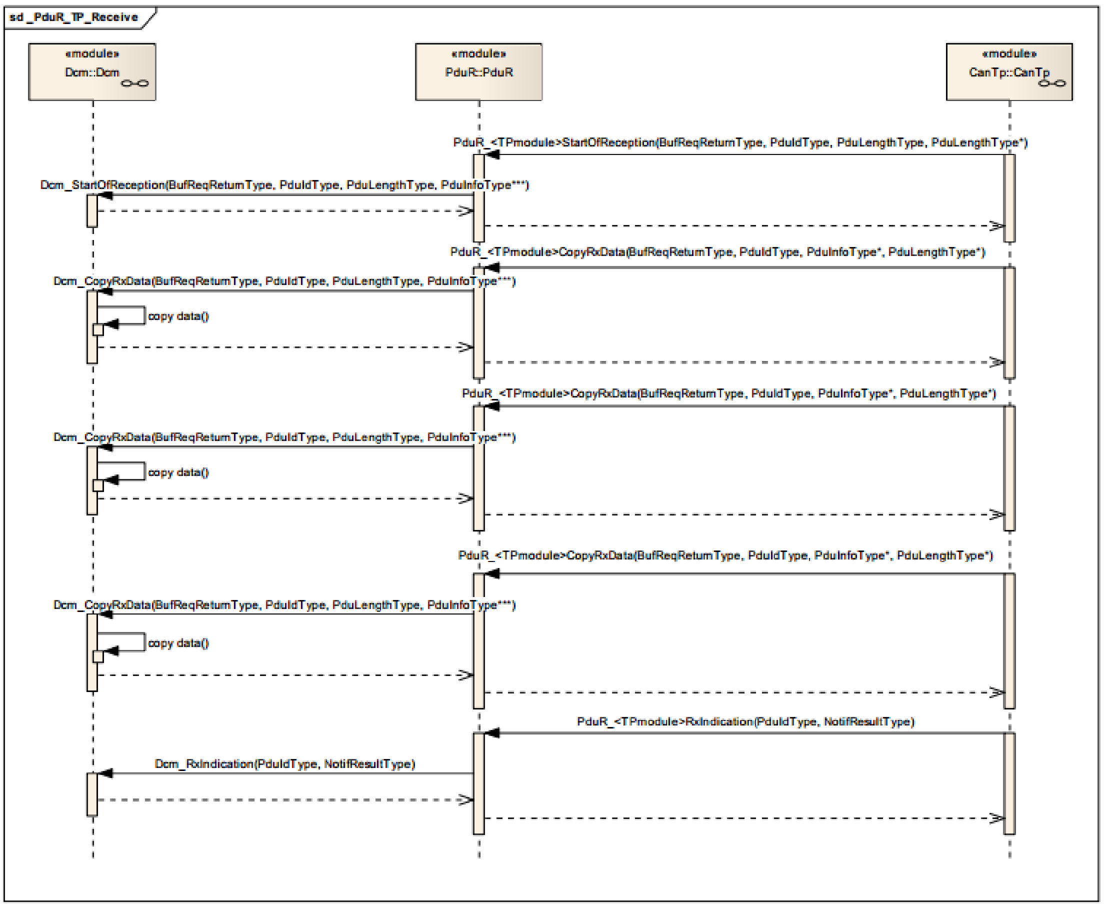

sd _PduR_TP_Receive
«module» Dcm::Dcm
«module» PduR::PduR
«module» CanTp::CanTp
PduR_<TPmodule>StartOfReception(BufReqReturnType, PduIdType, PduLengthType, PduLengthType*)
Dcm_StartOfReception(BufReqReturnType, PduIdType, PduLengthType, PduInfoType***)
PduR_<TPmodule>CopyRxData(BufReqReturnType, PduIdType, PduInfoType*, PduLengthType*)
Dcm_CopyRxData(BufReqReturnType, PduIdType, PduLengthType, PduInfoType***)
copy data()
PduR_<TPmodule>CopyRxData(BufReqReturnType, PduIdType, PduInfoType*, PduLengthType*)
Dcm_CopyRxData(BufReqReturnType, PduIdType, PduLengthType, PduInfoType***)
copy data()
PduR_<TPmodule>CopyRxData(BufReqReturnType, PduIdType, PduInfoType*, PduLengthType*)
Dcm_CopyRxData(BufReqReturnType, PduIdType, PduLengthType, PduInfoType***)
copy data()
PduR_<TPmodule>RxIndication(PduIdType, NotifResultType)
Dcm_RxIndication(PduIdType, NotifResultType)

Figure 26 - Communication between CanTp, Pdu Router and DCM modules. [12]

4.3.3 Diagnostic Sessions

After the primary bootloader has completed its initialization it will wait until a valid UDS message, Diagnostic Session Control or ECU Reset, is received or until the “start reprogramming” timeout timer has finished counting. The ECU is now in the default session and an overview of the paths that the bootloader program can take are shown in Figure 27. If the timer will time out the bootloader will start the ECU application stored in the memory by making a jump to a predefined memory address. If a Diagnostic Session Control message is received the timer be stopped and a transition to the received session will be made. The program flow will now on be completely controlled by incoming UDS requests. If there is no ECU application present in the memory then the bootloader will stay in the default session until Diagnostic Session Control message is received.

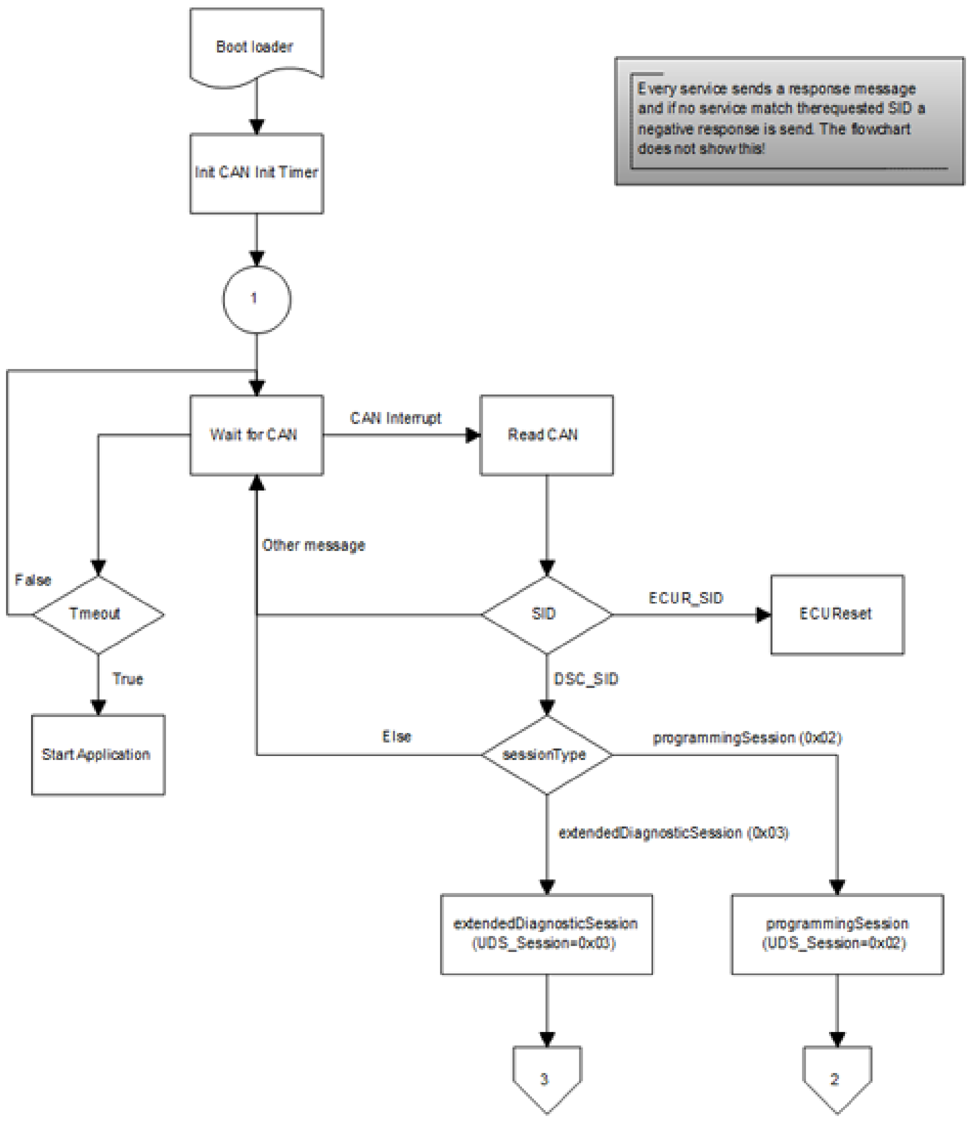

Boot loader
Init CAN Init Timer
1
Every service sends a response message and if no service match therequested SID a negative response is send. The flowchart does not show this!
Wait for CAN
CAN Interrupt
Read CAN
Other message
False
Tmeout
True
Start Application
SID
ECUR_SID
ECUReset
DSC_SID
Else
sessionType
programmingSession (0x02)
extendedDiagnosticSession (0x03)
extendedDiagnosticSession (UDS_Session=0x03)
programmingSession (UDS_Session=0x02)
3
2

Figure 27 - Program flow when the ECU is in the default session

In the Extended Diagnostic Session the program will wait for UDS requests and will stay in this session until a request to switch session or a reset message is received. The program flow can take two different paths, as shown in Figure 28, either execute Control DTS Setting and/or Communication Control or to switch to another session. If you choose to execute Control DTS Setting and/or Communication Control the program will stay in the extended diagnostic session after execution and wait for new UDS requests.

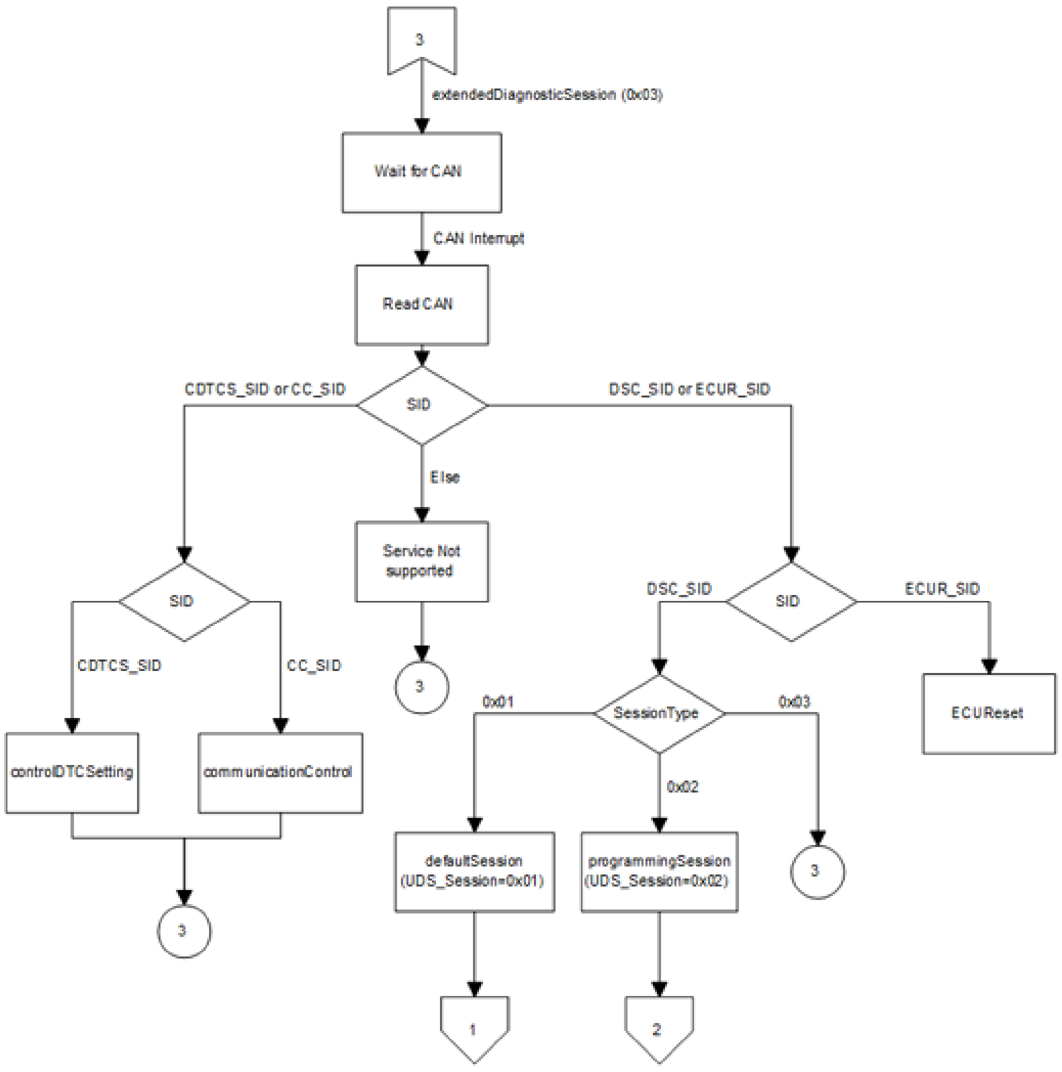


Figure 28 - Program flow when the ECU is in the extended session.

In the Programming Session the program can again take two different paths, execute functions in the programming session or switch to a different session with a Diagnostic Session Control message or by an ECU Reset message. To be able to execute any of the session specific functions, e.g. Request Download, they first have to be unlocked by executing the Security Access function. If access have been granted the programming sequence can begin when the first Request Download request message is received; see Figure 29.

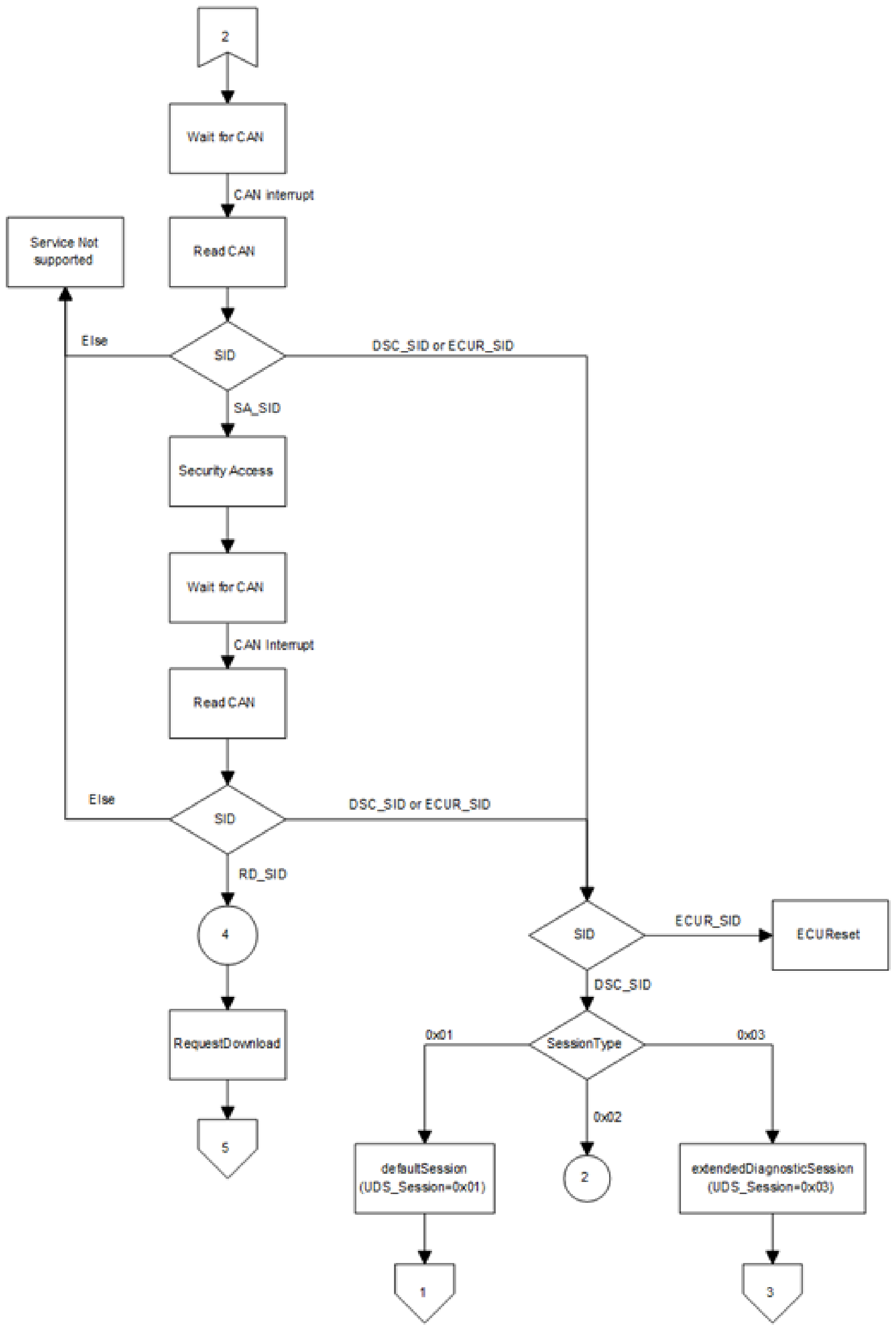


Figure 29 - Program flow when the ECU is in the programming session, part 1

To download data the following UDS request sequence has to be followed, Request Download -> Transfer Data -> Request Transfer Exit; see Figure 30. When a Request Download message containing data length and address have been received the the program is ready to receive the data and will wait for Transfer Data messages to arrive. If the data to be transfer is larger than what fits in one byte multiple Transfer Data messages will be sent after each other until all data have been transferred. When reprogramming of an ECU is to be performed the secondary bootloader will be downloaded in this step. The transfer will be ended when a Transfer Data Exit request message is received. The program is now ready to receive a new Request Download message and receive more data or to execute the Routine Control function; see Figure 30.

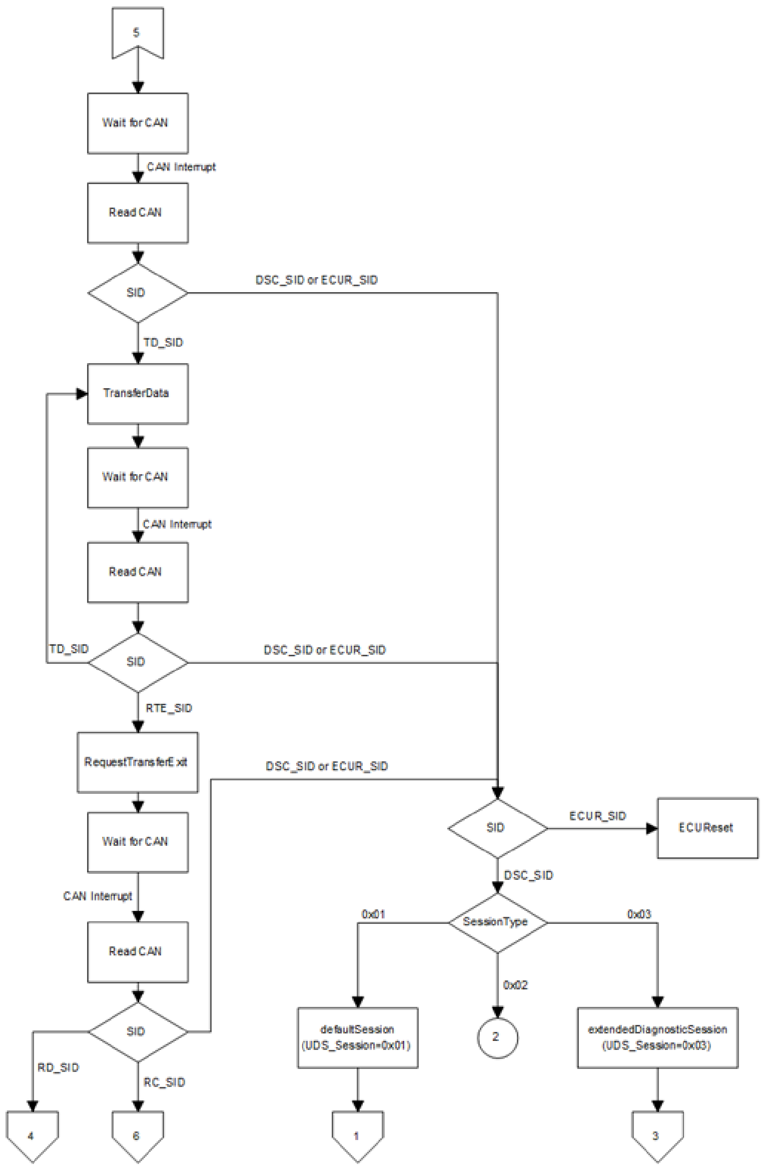


Figure 30 - Program flow when the ECU is in the programming session, part 2

The Routine Control function will execute the routine that are specified in the request. In Figure 31 the Routine Control starts a routine that was just downloaded before, in this example case the "Erase Flash" routine.

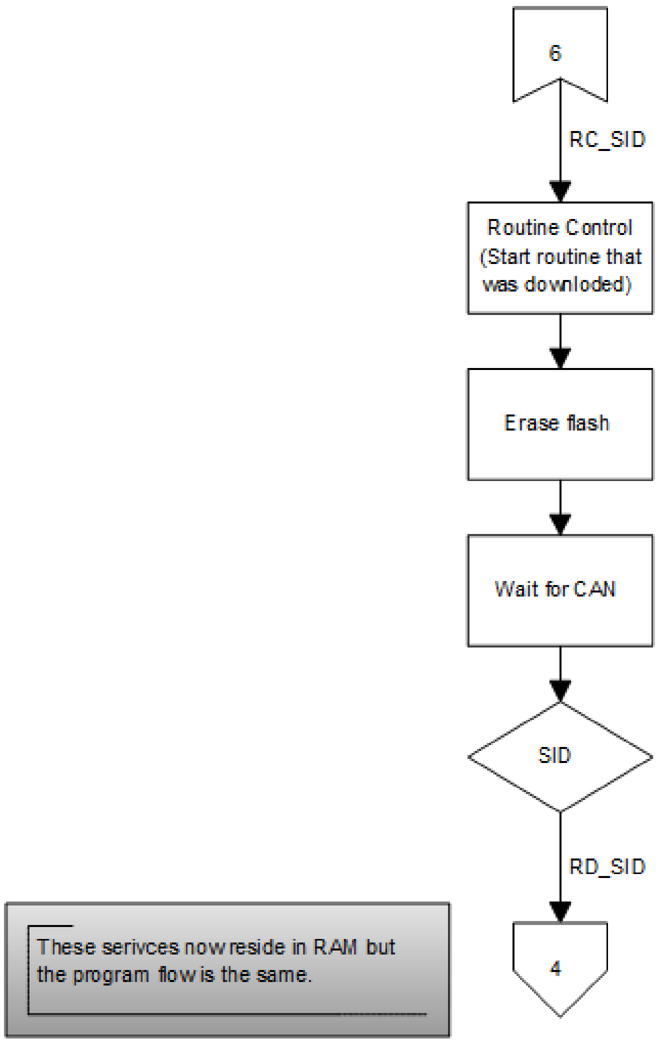

6
RC_SID
Routine Control (Start routine that was downloded)
Erase flash
Wait for CAN
SID
RD_SID
4
These serivces now reside in RAM but the program flow is the same.

Figure 31 - Program flow when the ECU is in the programming session, part 3

When the secondary bootloader has been downloaded and saved in the RAM memory the program will make a jump to the secondary bootloader after the Routine Control message have been received. The secondary bootloader will have the same program flow as have been described in the Programming Session. Default and Extended Diagnostic Sessions is not a part of the secondary bootloader. The secondary bootloader will receive the application to be stored in flash the same way as the secondary bootloader was received, Request Download -> Transfer Data -> Request Transfer Exit. When the application is download it is temporarily stored in RAM before it is written to the flash memory. Prior to storing the application the flash has to be initialized and unlocked if a writing lock is applied. Flash memory also has to be erased before new data can be written to it and both the erasure and the writing of data are verified before the reprogramming of flash is completed.

Because the part of flash reprogramming stayed in the development phase, in order to verify the proper operation of the diagnostic session flow according to UDS over CAN, a binary program (built as if it was the secondary bootloader) including the same function that blinks the LED but at a lower frequency was downloaded into SRAM in the target and properly executed afterwards.

4.4 Reprogramming

Since the implementation of the secondary bootloader was delayed and therefore not fully integrated with the primary bootloader, the flash reprogramming functionality was tested as a separate program loaded to run directly into RAM, by using the PEEDI device. This way, the functions corresponding to the flash driver proved to work well as it was possible to unlock, erase, write and verify flash memory blocks.

5 Discussion and conclusions

In this chapter the topics to be discussed include the judgment on the use of UDS and AUTOSAR for implementing the Primary and Secondary Bootloaders. Furthermore, a final word about the conclusions drawn from the results is given and also a short explanation is provided about which path this investigation could take to achieve a richer knowledgebase for comparing the method employed in the present work against those proposed.

5.1 Inconsistencies in AUTOSAR regarding operation of the bootloader

In *AUTOSAR Specification of MCU Driver* [36] a specification of the functions in charge of basic microcontroller initialization is provided. Such functions would be included in what is known as the MCU driver according to AUTOSAR. Furthermore a description of the guidelines for developing a so called “startup-code” is also given. It is stated that the following are the features of this MCU driver:

- Initialization of MCU clock, PLL, clock prescalers and MCU clock distribution
- Initialization of RAM sections
- Activation of µC reduced power modes
- Activation of a µC reset
- Provides a service to get the reset reason from hardware

The document then goes on and establishes that “*the initialization services allow a flexible and application related MCU initialization in addition to the start-up code*” (quote) and makes a reference to the picture shown immediately below.

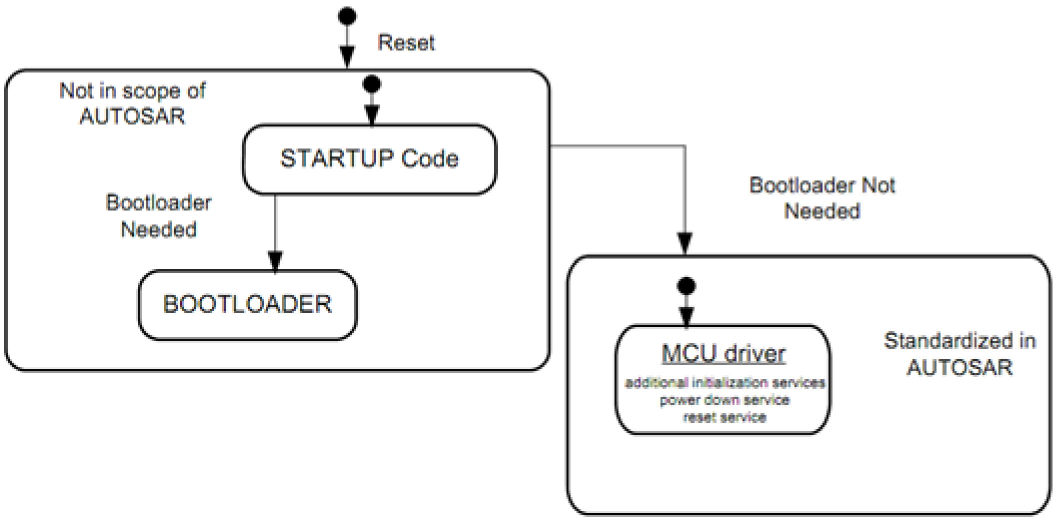


Figure 32 - Scope of the MCU Driver Specification [36]

As seen from the picture above, clearly the bootloader is considered a complete separate block of code that would run after the startup code has finished its operation and has nothing to do with the MCU driver.

From these statements and illustrations it can be seen that the concept of what a bootloader doesn't do (thus left to do by the MCU driver or startup-code) according to AUTOSAR doesn't hold to what it could be arguably considered the "common" model of a bootloader.

At first, some of the features of the MCU driver presented above actually match those a bootloader would include in its code such as initialization of MCU clock, PLL, prescalers and RAM sections. In addition, no reasons are given to explain when the Bootloader is needed and when it's not. A comparison with [12] helps to show these matching properties. For example, it is written there that the loader responsible for the bootstrap process "*initializes the RAM memory and possibly the on-processor cache*"(Li, 41) [1]. It is also explained there that part of the minimum hardware initialization within the boot sequence includes "*getting or setting the CPU's clock speed*" (Li, 48) [1].

Furthermore, the description of the startup code found on the AUTOSAR specification for the MCU driver points out quite well some functions regarding hardware initialization that more than one source agrees on that a bootloader should have. A very good example is on the statement that says: "*The startup code shall initialize the interrupt stack pointer (...) The interrupt stack pointer base address and the stack size are provided as configuration parameter...*"[37]. Similarly, in [12] it's explained that as part of the bootstrap process in order to put the system into a known state "the processor registers are set with appropriate default values. The stack pointer is set with the value found in ROM" (Li, 41) [1].

While it is a fact that the initialization process is very MCU-specific because of the different underlying hardware components, memory types and sizes, CPU's architectures and available resources for the programmer, still the bootloader can be recognized as the piece of code that takes care of such process. Because of reasons like those recently presented it is thought that the idea of what a bootloader is and does from the perspective of AUTOSAR is not concrete, unclear, avoided and instead spread among other pieces of code called "driver" or startup code, while it could well gather the functions these other codes provide and therefore become standard.

Going deeper into the inconsistencies, according to *AUTOSAR Specification of Diagnostic Communication Manager* [10] an implementation of the DCM that meets AUTOSAR requirements shall include "jumps" from and to a bootloader. However what a bootloader is and does, where it is located in memory and on which type of memory, when it starts and stops are aspects that aren't defined there neither referred to another supporting document whatsoever. Even a functional requirement is placed on the bootloader when it is stated that upon reception of UDS service LinkControl (0x87) the DCM realizes a jump to bootloader and the bootloader takes care of the LinkControl operation (setting a parameter for varying the baud rate of the communication interface). Nevertheless, it is never described whether the bootloader shall include the DCM with support for all possible UDS services, or just a separate subset of these without taking in count DCM's general functionality.

5.2 Results of implementing according to AUTOSAR

UDS consists of many services for different diagnostic functions. Only services that are needed to perform reprogramming of an ECU were implemented and all other services were left out. To delimit the thesis only the CAN communication interface was used for communication even though both the hardware and AUTOSAR supports other communication interfaces.

As a result of only have one communication interface, CAN, and only implement UDS, parts of AUTOSAR seems unnecessary, like for example PDU Router which only forwarded messages between CAN Tp and DCM. In a more complex system with multiple communication interfaces the purpose of PDU Router is clear but when there is only one module above and one below the need of routing is very limited.

5.3 Conclusions

Reprogramming will most likely be a part of AUTOSAR in the future but if it is done through UDS or XCP is hard to tell. Even though UDS have more of its functions needed for reprogramming already implemented today that does not mean that it will be the preferred method of choice in the future. Only through the comparison of practical implementations under the same testing conditions it will be possible to observe which standard offers better results in terms of code complexity, amount of overhead, reprogramming time and robustness, in case for example, the communication link is lost while transmitting/receiving data.

The complexity of the electronic systems in a car have increased rapidly over the last years and the manufacturers and Tier 1 suppliers are often faced with conflicting requirements from different stakeholders. Because of this, a need to standardized architectures and interfaces has arisen. Both AUTOSAR and UDS will be important parts of the automotive industry in the near future and the ability to easily upgrade the ECU software with new and improved functions will probably be an even more important feature in the future than it is today. Continuous work on AUTOSAR will unify the architectures in automotive software and this will make it easier for automotive manufacturers to combine technical solutions from different suppliers.

5.4 Further research

This part contains ideas how the research can be continued.

- Research reprogramming of ECU over different communication interfaces e.g. FlexRay or Ethernet.
- Investigate, implement and test the usability of other protocols, like XCP, for downloading software.

6 References

[1] Q. Li and C. Yao, "Real-Time Concepts for Embedded Systems," CMP books, 2003, pp. 35-50.

[2] S. Vidyadhara and A. Patil, "Best Practices in Boot Loader Design," Embedded Systems Conference – San Jose, 2006, pp. 3-9.

[3] "CiA," 2012. [Online]. Available: http://www.can-cia.org/index.php?id=systemdesign-can-history. [Accessed 11 05 2012].

[4] "Federal Register Environmental Documents," 2005. [Online]. Available: http://www.epa.gov/fedrgstr/EPA-AIR/2005/December/Day-20/a23669.htm. [Accessed 10 05 2012].

[5] "CAN Specification version 2.0," ROBERT BOSCH GmbH, Stuttgart, 1991.

[6] AUTOSAR, "Background," AUTOSAR, 2012. [Online]. Available: http://www.autosar.org/index.php?p=1&up=1&uup=1&uuup=0. [Accessed 11 05 2012].

[7] AUTOSAR, "Basics," AUTOSAR, 2012. [Online]. Available: http://www.autosar.org/index.php?p=1&up=1&uup=0. [Accessed 11 05 2012].

[8] AUTOSAR, "Layered Software Architecture 3.2.0," AUTOSAR, 2011.

[9] ISO, "ISO 14229-1 Road vehicles - Unified diagnostic services (UDS) Part 1," International Standards, 2007.

[10] AUTOSAR, "Specification of Diagnostic Communication Manager 4.0.0," AUTOSAR, 2009.

[11] AUTOSAR, "Specification of Diagnostic Event Manager 4.2.0," AUTOSAR, 2011.

[12] AUTOSAR, "Specification of PDU Router 3.2.0," AUTOSAR, 2011.

[13] AUTOSAR, "Specification of Communication Manager," AUTOSAR, 2011.

[14] AUTOSAR, "Specification of RTE," AUTOSAR, 2011.

[15] AUTOSAR, "Specification of Flash Driver 3.0.0," AUTOSAR, 2009.

[16] ASAM, "XCP General," in *XCP Part 1 Overview version 1.0*, 2003, p. 37.

[17] ASAM, "Absolute Access Mode," in *XCP Part 1 Overview version 1.0*, 2003, p. 38.

[18] ASAM, "Functional Access Mode," in *XCP Part 1 Overview version 1.0*, 2003, p. 39.

[19] AUTOSAR, "Specification of CAN Interface 4.0.0," 2009.

[20] GmbH, Vector Informatik, "Vector Katalog ECU Software," p. 118, 05 2012.

[21] AB, QRTECH, "ODEEP," [Online]. Available: www.odeep.se. [Accessed 11 05 2012].

[22] Freescale, MPC5567 Microcontroller Reference, 2007.

[23] IBM, "developersWorks Technical library," 30 03 2004. [Online]. Available: http://www.ibm.com/developerworks/linux/library/. [Accessed 01 01 2012].

[24] ISO, "ISO 15765-3 Road Vehicles - Implementation of Unified Diagnostic Services (UDS on CAN)," International Standard, 2004.

[25] ISO, "ISO 15765-2 Road vehicles - Diagnostic on Controller Area Network

(CAN) Part 2," International Standard, 2004.

[26] ISO, "Diagnostic Session Control," in *ISO14229-1 Road vehicles - Unified diagnostic Service (UDS) Part 1*, 2007, pp. 36-42.

[27] ISO, "Control DCT Settings," in *ISO14229-1 Road vehicles - Unified diagnostic Service (UDS) Part 1*, 2007, pp. 69-73.

[28] ISO, "Communication Control," in *ISO14229-1 Road vehicles - Unified diagnostic Service (UDS) Part 1*, 2007, pp. 52-55.

[29] ISO, "Security Access," in *ISO14229-1 Road vehicles - Unified diagnostic Service (UDS) Part 1*, 2007, pp. 45-52.

[30] ISO, "Request Download," in *ISO14229-1 Road vehicles - Unified diagnostic Service (UDS) Part 1*, 2007, pp. 231-234.

[31] ISO, "Transfer Data," in *ISO14229-1 Road vehicles - Unified diagnostic Service (UDS) Part 1*, 2007, pp. 237-242.

[32] ISO, "Request Transfer Exit," in *ISO14229-1 Road vehicles - Unified diagnostic Service (UDS) Part 1*, 2007, pp. 242-243.

[33] ISO, "Routine Control," in *ISO14229-1 Road vehicles - Unified diagnostic Service (UDS) Part 1*, 2007, pp. 225-231.

[34] ISO, "ECU Reset," in *ISO14229-1 Road vehicles - Unified diagnostic Service (UDS) Part 1*, 2007, pp. 42-45.

[35] AUTOSAR, "Specification of Module XCP," AUTOSAR, 2011.

[36] AUTOSAR, "Specification of MCU Driver 3.2.0," AUTOSAR, 2012.

[37] B. David, "TechRepublic," 17 06 2003. [Online]. Available: http://www.techrepublic.com/article/the-flash-rom-boot-loader-performs-critical-actions/5034893. [Accessed 11 05 2012].

[38] EE Times, "EE Times automotives," 12 08 2008. [Online]. Available: http://www.automotive-eetimes.com/en/in-system-programming-of-flash-via-control-unit-application.html?cmp_id=7&news_id=210002681. [Accessed 11 05 2012].