CODEN:LUTEDX/(TEIE-5370)/1-87/(2016)

Industrial Electrical Engineering and Automation

Automation of UDS-based flashing for software testing purposes in CANoe

Richard Pendrill

Division of Industrial Electrical Engineering and Automation
Faculty of Engineering, Lund University

Automation of UDS-based flashing for software testing purposes in CANoe

Richard Pendrill

LUNDS UNIVERSITET

Division of Industrial Electrical Engineering and Automation

Abstract

This Master's thesis investigates the possibility of adding full vendor-specific software loading sequence support to CANoe, in order to provide the possibility of testing the compliance of Electrical Control Units (ECU:s) from several different vendors to the international standard Unified Diagnostic Services ISO14229-1. Unified Diagnostic Services (UDS) specifies how diagnostic communication should be handled between a diagnostic tester and an on-vehicle ECU. This project was able to develop a framework for UDS-based software loading tests which could be run across several different ECU:s from BorgWarner PowerDrive Systems customers. This was achieved by focusing on creating a common format, downloading sequence and creating test cases which could be run across all projects based on the generic requirements identified in UDS.

Sammanfattning

Det här examensarbetet undersöker möjligheten att utföra fullständig tillverkar-specifik mjukvaruladdning i testmiljön CANoe för att kunna testa så att elektriska kontrollenheter (ECU:er) från flera olika tillverkare uppfyller de krav som finns i den internationella standarden Unified Diagnostic Services (UDS). UDS beskriver hur diagnostisk kommunikation skall hanteras mellan en diagnostik testare och en ECU. I det här projektet var det möjligt att skapa ett ramverk för automatiserade test på mjukvaruladdningssekvensen som kunde användas för test i flera olika ECU:er från BorgWarners kunder. Detta var möjligt genom att fokusera på att skapa ett gemensamt filformat och nedladdningssekvens för projekten hos BorgWarner samt utveckla testfall som kunde köras i alla projekt baserade på de generiska kraven som identifierats i UDS.

Acknowledgements

Firstly I want to thank Mattias Wozniak and Måns Andersson for the idea to this project, which fit perfectly as a Master's thesis as the amount of work, could be customized to a large extent. Mattias Wozniak has been the primary supervisor for this project; always answering every question I had and helped me structure the solution to this project by giving me relevant examples to follow and continuously discussing the structure. I also want to especially thank Marie Åkesson who provided relevant feedback continuously and together with the others in the team at the software-testing department made me feel welcome at BorgWarner PDS. Måns Andersson also deserves another mention for early on in the project involving me in a real customer issue giving me an idea of what test cases might be relevant investigating in the second part of the project. I also want to thank the rest of the team TTT-SW that also always took my questions seriously and helping out to their best ability when Mattias Wozniak was not available. The amount of support and advice I have received from the team at TTT-SW meant to some degree I did not need as much advice from my university supervisor Gunnar Lindstedt and examiner Ulf Jeppsson but it felt like if I would have needed more support from them I would have received it. Finally a mention of the developers at TTE at BorgWarner PDS who were a good source of information regarding what should be tested on the software loading sequence.

Table of Contents

Abbreviations

AUTOSAR – AUTomotive Open System Architecture
AWD – All-Wheel Drive
BIN – Binary format
BW-PDS – BorgWarner PowerDrive Systems
CAN – Controller Area Network
CAN-FD – Controller Area Network Flexible Data Rate
CANoe – Controller Area Network open environment
CAPL – CAN Access Programming Language
CIN – File format specific to CAPL for including functions and variables
CSMA/CA – Carrier Sense Multiple Access / Collision Avoidance
DoCAN – Diagnostic communication over Controller Area Network
DoIP – Diagnostic communication over Internet Protocol
DTC – Diagnostic Trouble Code
ECU – Electric Control Unit
EEPROM – Electrically Erasable Programmable Read Only Memory
ETC – European Tech Center
FXD – Front Differential Drive
GUI – Graphical User Interface
HEX – Intel hexadecimal file format
HIL – Hardware In Loop
ISO – International Organization for Standardization
ISO14229-1 – UDS: Specification and requirements – road vehicles
LIN – Local Interconnect Network
MOST – Media Oriented Systems Transport
NRC – Negative Response Code
OSI – Open Systems Interconnection
PBL – Primary Boot Loader
RAM – Random Access Memory
SBL – Secondary Boot Loader
SID – Service Identifier
SRE – Motorola S-format
STmin – SeparationTime minimum
TAE – Test Automation Editor
TDMA – Time Division Multiple Access
TTT-SW – Software testing department at BW-PDS
UDS – Unified Diagnostic Services
VBF – Versatile Binary Format
XCP – Universal Measurement and Calibration Protocol

1. Introduction

1.1 BorgWarner PowerDrive Systems

BorgWarner PowerDrive Systems (PDS) is a subsidiary of the American automotive industry component and parts manufacturer BorgWarner Inc. This thesis was done at the European Tech Center (ETC) in Landskrona, which specializes in torque transfer systems. The torque transfer systems developed and produced at ETC provide BorgWarner's customers with the ability to deliver on-demand all-wheel drive (AWD) or cross axle differential drive systems such as Front Differential Drive (FXD).

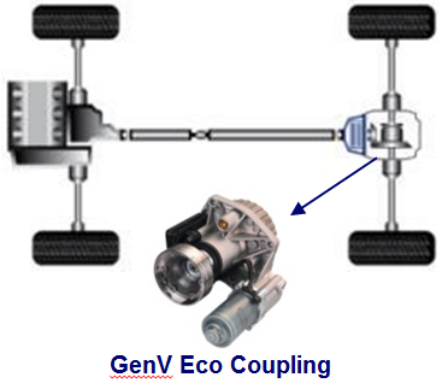

Figure 1-1: Generation V Eco coupling for providing all-wheel drive to a car [1].

These products provide improved fuel consumption and control compared to conventional AWD-systems. The Electrical Control Unit (ECU) controls the clutch, which regulates how and when the torque is applied to the different wheels of the car to achieve optimal traction when it is needed.

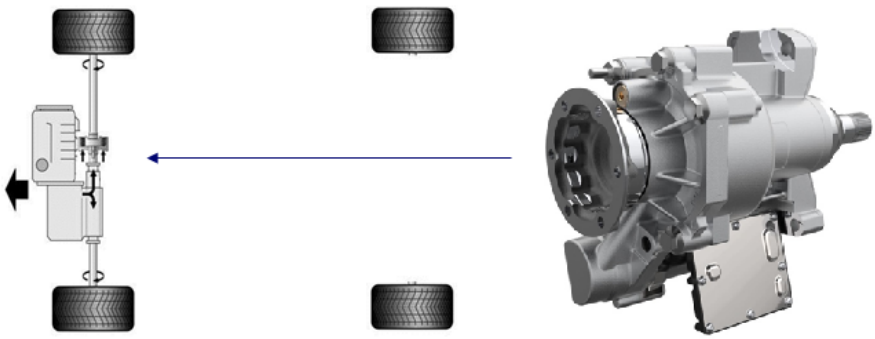

Figure 1-2: Front differential drive (FXD) coupling for vehicles as an alternative to a AWD-system [1].

1.2 Testing at BorgWarner

Before a product is delivered to the customer it has to go through a rigorous testing process to ensure to the greatest extent possible that problems do not arise during the life cycle of the product. Because automotive products have to handle both a long life-cycle and greatly varying operating conditions this requires a lot of attention and investment from a company like BorgWarner to ensure their reputation is not tarnished by a failure in any of their products. This thesis was done at the software-testing department at BW-PDS.

1.2.1 Software testing

The software-testing department at ETC handles the testing process for several different vehicle manufacturers, which supply consumers with automobiles with the torque transfer systems from BorgWarner PDS. Because the company supplies products to several different manufacturers there are great benefits of trying to have to the greatest extent possible a unified software testing strategy across all customer projects.

Therefore a lot of emphasis was placed in this project to get a testing process on the flashing sequence which was to the greatest extent possible unified and provide the possibility of running the same test cases on several different customer projects.

1.3 Problem formulation

In ongoing customer projects at BW-PDS some requirements specified in the ISO-standard "Road Vehicle Unified Diagnostic Services" (ISO14229-1, see Chapter 2.8) cannot be tested. Requirements that cannot be tested from their testing environment regards the software flashing procedure onto the ECU (See Chapter 2.7). Examples of this are: trying to transfer data, which is smaller or larger than specified, incorrect data and interruptions during different stages of the flashing procedure.

Today the software loading (flashing) of the ECU is performed by using an internally developed flashing tool (see Chapter 2.4). This flashing-tool is a standalone application, which is separate from the current test environment CANoe (see Chapter 2.5). This tool is also used in the production line; therefore the tool must be robust and easy to use.

In order to run test cases on the flashing procedure a way to control and supervise the flashing procedure has to be implemented into the current test environment. To be able to do the flashing procedure in the test environment a method to get the flashing data into CANoe has to be developed, as this is not possible today.

Questions to investigate are:

- How to read binary data into CANoe?
- BW-PDS customers have different formats of binary data, is it possible to create a generic format for all customers?
- Are there generic requirements and vendor-specific requirements in ISO14229-1 concerning the software loading sequence?
- Is it possible to write generic test cases that can verify the possible generic requirements identified in ISO14229-1?

1.4 Related work

The requirements on the software loading sequence specified in the UDS-standard create a need for testing these requirements in order to verify that the delivered ECU lives up to the specifications set by ISO14229-1. To accurately test these requirements a customizable software loading process is required which can simulate the variations of the software loading sequence, which are permitted in the UDS-standard. It is therefore likely that similar projects have been developed previously. One example of a similar solution to the one presented in this report is discussed in Chapter 1.4.2. At BorgWarner PDS a limited amount of the requirements have also been tested previously by using an internally developed

flashing tool discussed in Chapter 1.4.1. The internally developed flashing does however not provide the customizability of the flashing process that is needed in some tests.

1.4.1 Internally developed flashing tool

A tool developed internally at BW-PDS today performs the software loading. This tool is able to execute the vendor-specific software loading process for all BW-PDS customers. In order to support several vendors the flashing tool contains a number of functions for the conversion of vendor-specific binary data. These methods could be implemented into the conversion program, which provides the binary data to a software test environment.

1.4.2 Adding UDS over CAN to an HIL test system

A technical paper written by Matt Rings at National Instruments and Paul Phillips at Lear Corporation describes a similar problem formulation: ″*By adding Unified Diagnostic Services (UDS) over CAN to a Hardware-In-The-Loop (HIL) test system, Lear was able to increase test automation and provide wider test coverage by automating the ECU flashing process, adding diagnostic identifiers and trouble codes to their test scripts, and providing a quick and easy way to exercise ECU I/O.* [2]″.

The solution, which was implemented, utilized National Instrument (NI) tools VeriStand and LabView. The aim of this thesis is to provide the possibility of using ECU-testing specific tools from Vector Informatik GmbH such as CANoe, Test Automation Editor (TAE) and VTestStudio to run test cases on the ECU-flashing procedure. The main reason for using CANoe is that it is the software test environment, which is currently used at the software-testing department at BW-PDS. Another advantage with CANoe is the possibility to write module-based code so that large parts of the functionality can be platform independent. In theory the only platform specific module needed would be the actual transmission of the data.

1.5 Outline of report

Chapter 2 in this report aims to cover the fundamental background needed to understand how the implementation of this project is carried out and provide context of how the UDS-standard is formulated to perform a software loading sequence. In Chapter 3 the equipment needed to implement this project is briefly discussed. Then Chapter 4 covers the implementation of getting the binary data

into CANoe, performing vendor-specific software loading sequence and finally the structure of the flash tests, which were run in this project. Chapter 5 handles the final results from this project, how results are presented to the test engineer utilizing this project for flash testing. Chapter 6 discusses what has been learned from this project, what could be possible future improvements and work.

2. Background

This chapter aims to cover the background needed to add context and understanding of how this project was developed. First the communication buses in automotive applications are discussed in Chapter 2.1, which are the currently used communication buses and how this might change in the future. Then the formats, which are used for transporting the binary data by BW-PDS's customers for the software loading process, are presented in Chapter 2.2. The non-volatile memory types are briefly discussed in Chapter 2.3, which are typically used in standard ECU:s to store the software for controlling the coupling in the torque transfer system. The process of performing a flashing sequence with the internally developed flashing tool is discussed in Chapter 2.4. A large part of this chapter is dedicated to discussing the ISO-standard Unified Diagnostic Services in Chapter 2.8 to give context to the implementation and analysis of the automatic test cases that were implemented in this project. Other topics which are discussed in this chapter include ECU:s, the software tools from Vector Informatik GmbH that made this project possible and the AUTOSAR initiative to promote standardization in the automotive field.

2.1 Communication buses in automotive applications

Communication networks in automotive applications have special requirements, which have to be fulfilled: they have to be able to operate in harsh operating environments, resistant to external disturbances, cost efficient and provide reliable and robust communication, which satisfies real-time communication requirements in vehicles. In order to fulfill these requirements dedicated fieldbuses are used in automotive applications.

In recent years the communication networks in the automotive sector have grown larger as more and more mechanical systems in cars are being replaced by

electronic systems. This development is likely to continue, as there are considerable benefits of being able to control the different components in cars more precisely. Examples of the improvement, which can be achieved, are reducing exhaust emissions and provide more features to consumers in order to compete in a global market.

The dominating communication protocol used in automotive is CAN or Controller Area Network. The main reasons for this are its cost effectiveness and because it is a tried and tested communication protocol, which was first introduced in 1980:s. However, as the amount of data increases and new features that require deterministic behavior such as driver assistance features, there is a growing need for new communication protocols.

Examples of new communications protocols are FlexRay, MOST and LIN each with their own advantages and disadvantages. A modern automobile will typically use several of these communication standards in order to balance performance and cost.

There is also a new version of CAN being developed by Bosch, which is called CAN FD. CAN FD seeks to address the problem of low bandwidth associated with the CAN-bus [3] [4].

2.1.1 Controller Area Network

Bosch developed Controller Area Network or CAN in 1980:s for fast serial data communication in automotive applications. Decades later it is still the dominating communications network used in cars. The CAN-bus has been able to remain the primary standard used in vehicles even as the amount of data transmitted has increased. This is because of the fact that vehicle manufacturers could simply add additional CAN-networks, rather than increasing the bandwidth on a single network. The main properties that were included in the CAN-bus in the 1980:s continue to be as relevant today: resistance to external disturbances in an automotive environment and important real-time characteristics.

Network topology of a CAN-network

The network topology of a CAN-network is depicted in Figure 2-1. All CAN-nodes on a CAN-network have permission to access the CAN-bus at any time (Multi-master bus). Each node has its own controller and transceiver, which enables the multi-master decentralized structure of the CAN-bus. Two important CAN-protocol features, which enable the important real-time characteristics of the CAN-bus, are CSMA/CA and an arbitration scheme discussed on the next page.

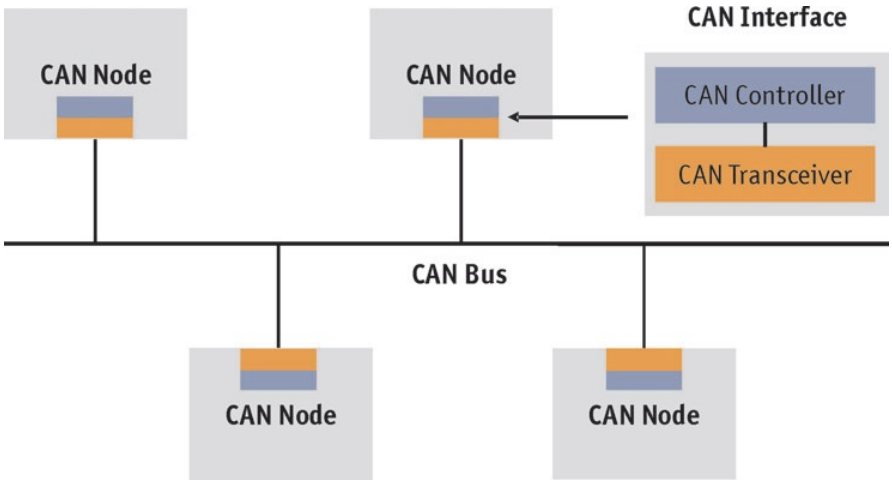

Figure 2-1: Example of the CAN-network topology, each CAN-node in the network has their own controller and transceiver. Each node has permission to access the CAN-bus without earlier coordination with the other CAN-nodes on the bus. Figure from [5, p. 2].

CSMA/CA and Arbitration Scheme

CAN is an event driven communications protocol, which enables quick reaction time to events. In order to maintain reliable real-time behavior, even when there are multiple CAN-nodes operating on the same bus, two important techniques for bus access are used: firstly Carrier Sense Multiple Access / Collision Avoidance or CSMA/CA and secondly an arbitration scheme.

When a CAN-node has a message to send it first checks if there is another node already communicating on the bus. If there is no ongoing communication the node sends its message on to the bus. If the CAN-bus is occupied then the node waits a specified amount of time before checking again if the CAN-bus is available. If there is a collision in spite of the CSMA/CA-technique then the arbitration scheme is used to prioritize the most important message on the bus. An example of this arbitration scheme is shown in Figure 2-2.

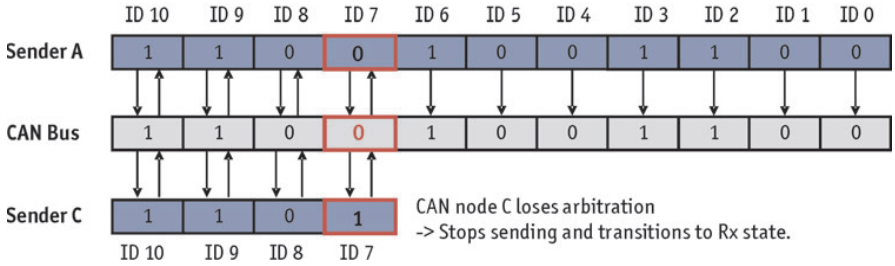

Figure 2-2: Arbitration scheme used in CAN-communication. When multiple nodes are sending messages on the CAN-bus simultaneously the message with the highest priority(0 - high priority, 1 - low priority) will continue sending its message, whereas the lower priority message will stop transmitting. In this example sender A has the higher priority message and therefore retains access of the CAN-bus. Figure from [5, p. 3].

2.1.2 Controller Area Network FD

The main limitation of the traditional CAN-network is the restricted bandwidth, which is a maximum of 1 Mbit/s on a 40 meter CAN-bus. CAN-Flexible Data-Rate (CAN-FD) is a new standard developed by Bosch aiming to increase the bandwidth of the CAN-bus while retaining the core characteristics of the traditional CAN-bus. The bandwidth restriction in CAN arises due to the arbitration scheme illustrated in Figure 2-2, which is used in the CAN-standard.

On a CAN-bus multiple nodes are allowed to transmit at the same time before the higher priority message wins the arbitration as the sender A in Figure 2-2 does at ID 7. In order for the arbitration scheme to work for all communication on the bus, the signals from a node must be able to propagate through the entire length of the CAN-bus and back again. In order to ensure corresponding bits are compared in the arbitration scheme, even for the nodes, which are furthest away from each other. However once the higher priority node gains access to the CAN-bus only one node will be transmitting data. By utilizing this characteristic of the CAN-bus it is possible to transmit data at a higher rate once there is only one node transmitting on the bus. This is the main idea behind the new CAN FD standard [3] [5].

2.1.3 FlexRay

FlexRay is a relatively new communications bus, which is expected to replace the CAN-bus for high-end applications in future automobiles. The main advantages of the FlexRay-bus over the CAN-bus are higher bandwidth and support for

deterministic communication. A consortium of automobile- and semiconductor manufacturers developed FlexRay that became an ISO-standard in 2010.

Network topology of a FlexRay-network

The FlexRay standard adds flexibility to the design process of the network topology compared to the CAN-standard. In addition to supporting multi-drop bus topology, which is used in a CAN-network, the FlexRay-protocol also supports a star network topology. In a star network topology there is a central node which handles the communication between the different ECU:s.

Figure 2-3: Star network topology, a central node handles the communication, similar to a switch or router in a PC Ethernet network. Figure from National Instruments [4, p. 2].

The network topologies can also be mixed in order to create an optimized network in which the advantages and disadvantages of the two network topologies are utilized.

Communication cycle and time division multiple access

A FlexRay network can accommodate both efficient data transfer and deterministic behavior when required. The FlexRay standard manages communication on a single bus with multiple nodes by using a Time Division Multiple Access (TDMA) scheme. ECU:s have a predetermined timeslot where they are permitted to transfer data in a communication cycle. The communication cycle in the FlexRay protocol is divided into different parts, there are static and dynamic segments. There is also a symbol window and idle timeslot. The structure of a communication cycle can be seen in Figure 2-4.

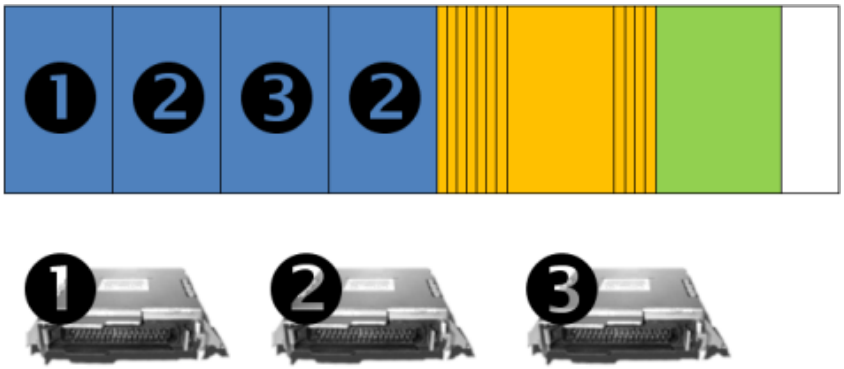

Figure 2-4: Communication cycle for a FlexRay network. The communication cycle contains different parts: a static segment (blue), a dynamic segment (yellow), a symbol window (green) and an included idle time (white). Figure from National Instruments [4, p. 3].

In a static segment a timeslot is reserved for one ECU, in which only the specified ECU may transmit data as seen in Figure 2-4 where ECU #1 and #3 have one determined timeslot in the static segment (blue) and ECU #2 has access to 2 timeslots to transmit data. The static ensures deterministic communication, which is important in various applications. It is for example important when calculating control loops where equally spaced measurements are advantageous.

In the dynamic segment the communication is divided into micro-slots where each ECU has the ability to signal that it has data to transmit to the network. The ECU:s with the highest priority will have a micro-slot earlier in the dynamic segment in order to ensure that the ECU:s, which have higher priority will get access to the FlexRay bus before lower priority ECU:s. Once an ECU has received permission to send it will occupy the bus. The dynamic segment in the FlexRay bus has similar real-time characteristics to the communication in a CAN-network. The mix of static and dynamic segments makes it possible to achieve real deterministic behavior and at the same time not sacrificing the performance of the network by permitting all ECU:s to occupy a timeslot in the dynamic segment when the ECU has data to send.

The symbol window is mainly used when performing a startup of a FlexRay network and the idle timeslot is used to synchronize all ECU-nodes to the communication cycle so that all nodes will communicate at the correct timeslot in the communication cycle [4] [6].

2.1.4 MOST and LIN

Local Interconnect Network or LIN was developed to support applications where the features of the CAN-bus were unnecessary and therefore more expensive than what they had to be. The LIN standard is today used in applications where the relatively high data transfer rate and robust characteristics of the CAN-bus are not required. Examples of these applications are seat, door and mirror control as well as climate control in the vehicle.

Media Oriented Systems Transport or MOST is another communications standard used in automotive applications. It is optimized for multimedia and infotainment applications that require high data transfer rates [7] [8].

2.2 Formats of binary data

The binary data for the software loading sequence is supplied by the different vehicle manufacturers to BW-PDS. The main formats used by BW-PDS customers are Versatile Binary Format (VBF), Motorola S-format (SRE) and binary format (BIN). Another format that is commonly used delivering binary data is the Intel HEX format.

2.2.1 Motorola S-format

The binary data contained in the SRE-format is represented according to the Motorola S-format. Motorola created the Motorola S-format for the Motorola 6800-series 8-bit microprocessors in the 1970:s. The main benefit of the S-format is that it provides a way to transport data in a way, which can be visually inspected.

The S-format file contains several lines of S-records, each S-record contains five data fields: record type, byte count, address, binary data and checksum. In Figure 2-5 a typical S-format file is depicted. The S0-record is the header record for all S-format files and the S7-record is a termination record for a S-format file with S3 data records. The S3-record contains 4 address field bytes. The address field of the first S3 data record is used in the internally developed tool (see Chapter 2.4) to identify which file block is being read. There are a few different record types specified in the Motorola S-format standard with varying address field size. The byte count is denoted in hexadecimal, in this case the S3-record is 37 bytes long (25 in hex), included in this number the address field therefore there are 33 data bytes in this record (37 bytes in total – 4 address field bytes = 33 data bytes in total) [9] [10] .

```
S0140000286329205441534B494E472C20496E632E7272
S32500C0B400     Data bytes, 33 bytes in total...     2E42
S32500C0XXXX          More S3 records.....           3A42
S32500C0XXXX          More S3 records.....           0042
S32500C0XXXX          More S3 records.....           0042
S7140000286329205441534B494E472C20496E632E7276
```

Figure 2-5: A typical Motorola S-format file used to deliver the binary data to the software loading process. There are three S-records primarily used in the S-format files from BW-PDS customers. A header record S0, a data record S3 with 4 bytes address information and a termination record S7.

2.2.2 Versatile Binary Format

The Versatile Binary Format or VBF is a format used by Volvo Car Corporation and its collaborating partners [11]. The file format contains a header that contains useful information for the software loading sequence in addition to the binary data.

2.2.3 BIN-format

In the BIN-format there is only the binary data without a structure, therefore it is not possible to easily visually inspect the data as can be done with the Motorola S-format or Intel HEX. Furthermore, the format does not contain all information needed in the software loading sequence (see Chapter 2.4.2 how this is solved in the internally developed flashing tool).

2.2.4 Intel HEX format

A binary file format that is frequently used when programming or flashing microcontrollers is the Intel HEX format. In many ways it is similar to the Motorola S-format. Containing the same kind of fields as the S-format: Fields for record type, byte count, address field, binary data and a checksum for each record.

A standard Intel HEX file format is depicted in Figure 2-6.

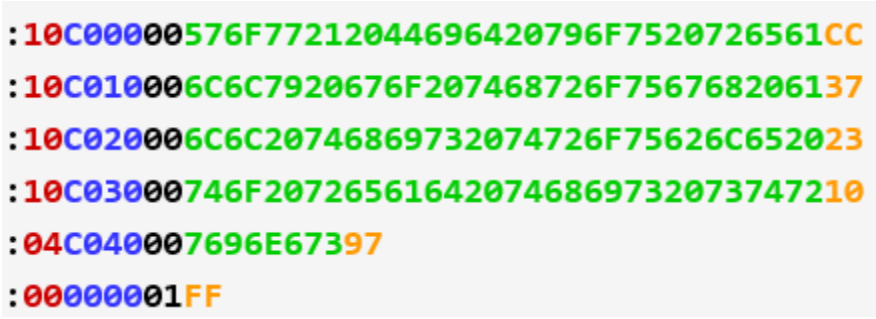

Figure 2-6: A standard Intel HEX format binary data file. All records start with a colon ":", byte count (Red), address field (Blue), record type (Black), binary data (Green) and a checksum (Yellow) on each record. Figure from SB-Projects [12].

The record types most frequently used are “00” which is a standard data record and “01” which is an end of file record [12].

2.3 EEPROM and flash memory

Electrically Erasable Programmable Read-Only Memory or EEPROM is a non-volatile memory, which is used in some hardware implementations of ECU:s (see Chapter 2.7). A special feature of the EEPROM-memory is that it typically allows modification of individual byte values whereas other non-volatile memory types typically only allow modifications of entire byte blocks. One example of a non-volatile memory where only entire byte blocks may be modified is flash memory, which is a memory type which has become increasingly popular as prices and performance of the flash memory have significantly improved in the last years. Flash memory was originally developed from EEPROM and is today extensively used in solid-state drives (SSD), memory cards and USB-sticks [13] [14].

2.4 Internally developed flashing tool

The software loading to ECU:s at BW-PDS is done by an internally developed tool. The tool has two Graphical User Interfaces (GUI:s) to support all the different projects at BW-PDS. The main GUI can be seen in Figure 2-7. The separate user interface for loading BIN-files can be seen in Figure 2-8.

2.4.1 Normal process for performing a flash

The normal process for using the tool is that the user chooses the project, which is going to be flashed under the dropdown menu “Flasher”, then adds the files needed for the flashing sequence with the “Add file to list” button. Finally the software loading sequence is started by pressing the “Download” button. There is a progress bar tracking the flashing process and any error during the process is printed out to the small white box just above the “Download” button. There is also a “radio button” for choosing which CAN-channel (Channel 1 or 2) the tool should send the required UDS-messages (see Chapter 2.8) for the flashing sequence.

There is an option to save the current configuration in a file, which can be opened the next time a flashing has to be performed; thereby the user only has to configure the flashing process when there is a new software release.

2.4.2 Loading BIN-files with the separate interface

As the BIN-files do not contain an identifier in the binary data it is not possible to separate the file blocks from each other based on the data contained in the file. Therefore a separate interface is needed for the projects that have their binary data delivered in BIN-files. The user adds the binary data files to each row with the corresponding block number (Block NR) in order to add an identifier to each file block. A separate signature file containing checksums for each block is loaded automatically if it is available in the same folder. When the user is finished with the binary file inputs the user presses the “save and return” button returning to the main graphical user interface.

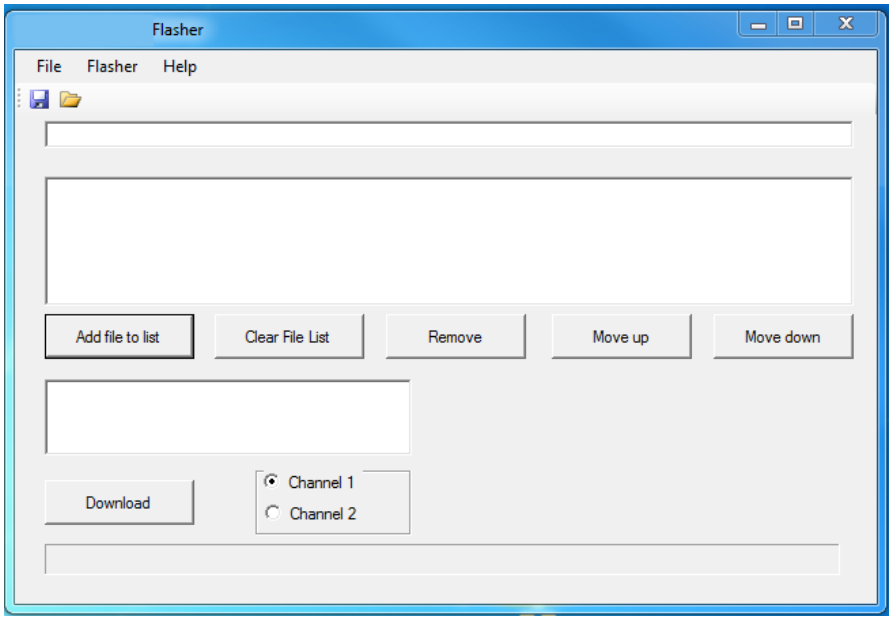

Figure 2-7: The main graphical user interface (GUI) for the internally developed flashing tool. The user adds the binary data files for the flashing.

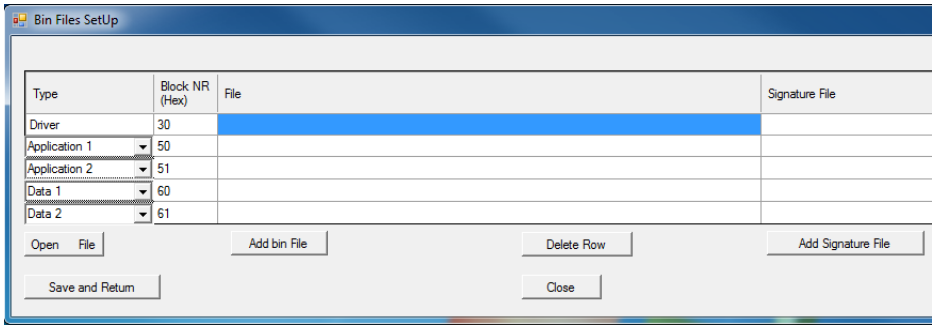

Figure 2-8: The separate user interface for loading BIN-files. The user adds the binary data files to each row with the corresponding block identifier (Block NR). A signature file is loaded automatically if it is available in the same folder.

2.5 CANoe

CANoe is a software testing tool developed by Vector Informatik GmbH, which can be used for development, testing and analysis of individual ECU:s or entire networks of ECU:s.

It provides the possibility to simulate how the ECU network would behave in an actual car so that the functions of the ECU can be tested in a simulation of the environment in which it will later be used. This makes it possible to test ECU:s in a realistic environment without having an actual car present in the testing process [15].

2.6 CAPL

CAN Access Programming Language or CAPL is the programming language used in the Vector-based programs CANoe and CANalyzer. CAPL is based on the C-programming language but adds features to support CAN-based embedded systems development.

CAPL is an event driven programming language. CAPL applications can be developed to respond to different system events such as key press, software timers, CAN-messages, and then executing a routine in an interrupt-like manner.

There are several built in functions in CAPL for diagnostic communication that will be extensively used in this project for the UDS-communication between the test environment and the ECU during the software loading process.

There are two different file formats supported by CAPL: CIN- and CAN-files [16].

2.6.1 CIN and CAN-files

A CIN-file is a non-executable format used in CAPL, which can include functions, constants and variables. Functions and variables that can be used in several different applications are typically stored in CIN-files. This enables reuse of commonly used variables and functions. The CIN-files are included in a CAN-file, which will then gain access to the functions and variables included in the CIN-files. A simple example of this is shown in Figure 2-9, which depicts a simple CIN-file. Figure 2-10 and Figure 2-11 depict two independent CAN-files which both utilize the variable included in the CIN-file.

```
File Simparameter.CIN ::
variables
{
  int _SimPar_Granularity_ms    = 10;
}
```

Figure 2-9: A simple CIN-file, which includes a variable "_SimPar_Granularity_ms". Example file from Vector Informatik GmBh, Stuttgart, Germany [17].

```
File Door_Left.CAN ::
includes
{
  #include "Simparameter.CIN"
}

variables
{
  msTimer cyclicTimer;
}

on start
{
  setTimer(cyclicTimer, _SimPar_Granularity_ms);
}

  on Timer cyclicTimer
{
  // Main simulation loop Door left
  setTimer(cyclicTimer, _SimPar_Granularity_ms);
}
```

Figure 2-10: A simple CAN-file, which includes the CIN-file "Simparameters.CIN" in Figure 2-9. It is then able to use the variable included in the CIN-file. Example file from Vector Informatik GmBh, Stuttgart, Germany [17].

```
File Radio.CAN ::
includes
{
  #include "Simparameter.CIN"
}

variables
{
  msTimer cyclicTimer;
}

on start
{
  setTimer(cyclicTimer, _SimPar_Granularity_ms);
}

on Timer cyclicTimer
{
  // Main simulation loop Radio
  setTimer(cyclicTimer, _SimPar_Granularity_ms);
}
```

Figure 2-11: Another simple CAN-file that includes the CIN-file "Simparameters.CIN" in Figure 9. It is then able to use the variable included in the CIN-file. Example file from Vector Informatik GmBh, Stuttgart, Germany [17].

2.7 Electric Control Unit

An Electric Control Unit or ECU is an embedded computer system, which controls parts of the electrical system in a motor vehicle. By gathering and processing information from several sensors (for example temperature sensors, accelerometers and gyroscopes) placed in different parts of the vehicle it can control various automated processes in the vehicle. ECU:s are also used to check performance of key components in the car and to monitor changes over time [18].

2.7.1 ECU architecture

A reprogrammable ECU has to have a hardware and software architecture, which supports the software loading process. An ECU, which is reprogrammable after it leaves the manufacturing plant, has the benefit that fixes and new features can be added later during the lifetime of the vehicle. There are also downsides to having a reprogrammable ECU: by enabling more people to alter the software on the ECU, some control of how the software loading is done is lost. Because of this the ECU must implement a structure, which prevents the software on the ECU from becoming unusable. In order prevent the ECU from becoming unusable there is usually a protected flash memory or EEPROM sector (see Chapter 2.3) where a

primary bootloader or PBL is placed. The primary bootloader should theoretically be impossible to remove without special access, and should not be altered during a normal software loading sequence, see memory structure of a typical reprogrammable ECU in Figure 2-12 [11].

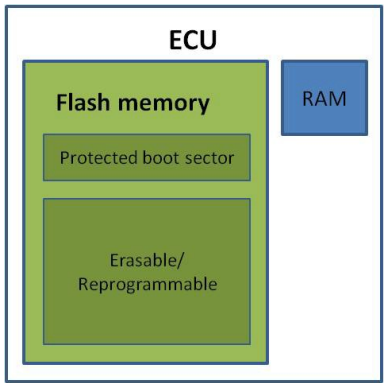

Figure 2-12: Memory structure of a typical programmable ECU. The flash memory or EEPROM contains a protected sector where the primary bootloader (PBL) is stored. Figure by V. Bordyk [11, p. 5].

When the ECU is powered up the primary bootloader will the first code that will be run. The primary bootloader will then start the application on the ECU if there is a valid application installed (see Figure 2-13). The principle of using a bootloader is ubiquitous in computer systems, for example in a normal desktop PC a bootloader will be run before the operating system is loaded.

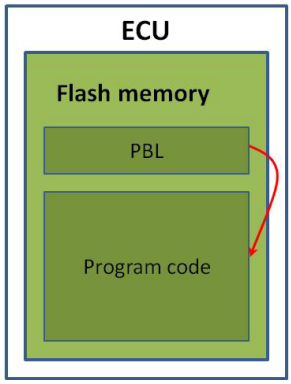

Figure 2-13: When the ECU is powered up the primary bootloader (PBL) is run first, the PBL will then start the application if there is a valid application loaded. Figure by V. Bordyk [11, p. 6].

In all software loading processes at BW-PDS the ECU:s use a secondary bootloader (SBL) to write new application and parameter data files. The PBL supports the downloading of the SBL to RAM, then the SBL will control the writing and erasing of flash- or EEPROM memory (see Chapter 2.3) for application and parameter data files. Because the SBL is placed in RAM, which is a volatile memory type, it will not be available after the ECU has been reset or turned off.

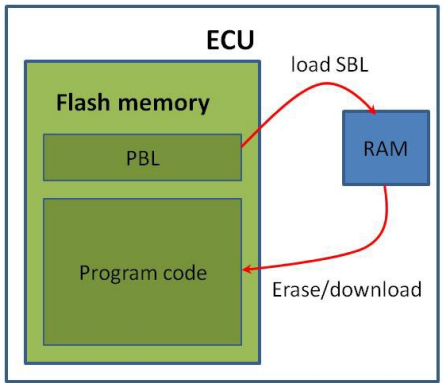

Figure 2-14: The secondary bootloader (SBL) is downloaded to the RAM by the primary bootloader (PBL). The SBL will then support the software loading of the application and parameter files. Figure by V.Bordyk [11, p. 6].

2.8 Unified Diagnostic Services ISO14229-1

The ISO-standard UDS defines how diagnostics communication should be handled between a diagnostic tester (client) and an on-vehicle ECU (server). The UDS-standard requires that several diagnostic control functions should be available independent of data link.

There are general message conventions in the UDS-standard; the most important ones for this project are discussed in Chapter 2.8.2.

During the software loading (flashing) of the ECU several of the diagnostic services described in UDS are used. The most important services utilized in the flashing of the ECU are listed in Table 2-1 and explained more in detail how they are used in the flashing sequence in Chapter 2.8.3. Then the steps of a standard UDS software loading sequence are discussed in Chapter 2.8.4. This entire chapter contains information gathered from the ISO14229-1 standard and aims to present the information, which is important for a standard software loading sequence and test cases on this sequence [19].

2.8.1 Overview of software loading services specified in UDS

Downloading new software to an ECU is generally done in a similar way independent of which vehicle manufacturer specific sequence is used. The UDS-services that are frequently used in the software loading sequence are listed in Table 2-1:

Table 2-1: Important software loading services specified in UDS

Identifier Byte (SID) #1	**Sub-function/data-parameter#1** Byte #2	Description of services used in software loading.
0x10	**DiagnosticSessionControl** (0x01) defaultSession (0x02) programmingSession (0x03) extendedDiagnosticSession (0x04) safetySystemDiagnosticSession (0x40-0x5F) vehicleManufacturerSpecific	Enables selection of different sessions. The most important sessions are: defaultSession, programmingSession and extendedDiagnosticSession. Some vehicle manufacturers have specific sessions (0x40-0x5F).
0x11	**EcuReset** (0x01) hardReset	Hard Reset simulates a start-up sequence after an ECU has been disconnected from its power supply (i.e. battery).
0x27	**SecurityAccess** (0x01)/ (0x00-FF) requestSeed (0x02)/ (0x00-FF) sendKey	All reprogrammable ECU:s should restrict access from unapproved tools. This service is used to unlock ECU before downloading and uploading of data.
0x85	**ControlDTCSetting** (0x01) On (0x02) Off	Used to stop or resume updating of Diagnostic Trouble Code (DTC) status bits on the ECU.
0x28	**CommunicationControl** Byte #2: (0x00) EnableRxAndTx (0x01) EnableRxAndDisableTx. (0x03) DisableRxAndTx Byte#3: (0x01) NormalCommunication	Service used to control transmission of normal communication. For example: disable or enable transmission of all non-diagnostic communication on an ECU.
0x22	**ReadDataByIdentifier** (0x00-FF) dataIdentifier (Can be specified in several bytes)	Service to read data at location specified by dataIdentifier. For example reading ECU assembly number, serial number and software identification number.

0x2E	**WriteDataByIdentifier** (0x00-FF) dataIdentifier (Can be specified in several bytes)	Service to write data at location specified by dataIdentifier. For example writing ECU assembly number, serial number and software identification number.
0x31	**RoutineControl** Byte #2: (0x01) startRoutine, (0x02) stopRoutine. (0x03) requestRoutineResults Byte #3&4: (0x00-FF) routineIdentifier	Service to execute a defined sequence of steps. Flexible service which is for example used to start download to RAM, erasing flash memory and for calculating checksums. routineIdentifier specifies routine.
0x34	**RequestDownload** Byte #2: (0x00-FF) dataFormatIdentifier Byte #3: (0x00-FF) addressAndLengthFormatIdentifier	Used to initiate a data transfer from the tester to the ECU. dataFormatIdentifier is used to specify if data is compressed and/or encrypted.
0x36	**TransferData** (0x00-FF) blockSequenceCounter	Service to transfer data from tester to ECU. blockSequenceCounter is included to allow for error handling if an error occurs during sending of multiple Transfer Data requests.
0x37	**RequestTransferExit**	Used to terminate data transfer between tester and ECU.
0x3E	**TesterPresent**	This service is used to inform the ECU that a tester is still present and therefore should remain in current session.

2.8.2 General UDS-message conventions

There are general service description conventions in the UDS-standard of how a request, positive- and negative responses should be formulated. A standard request is structured according to Figure 2-15. Each UDS-service has its own request identifier or request service identifier (SID, see Table 2-1). Byte #2 will specify either a sub-function of the UDS-service or the first data parameter in the request.

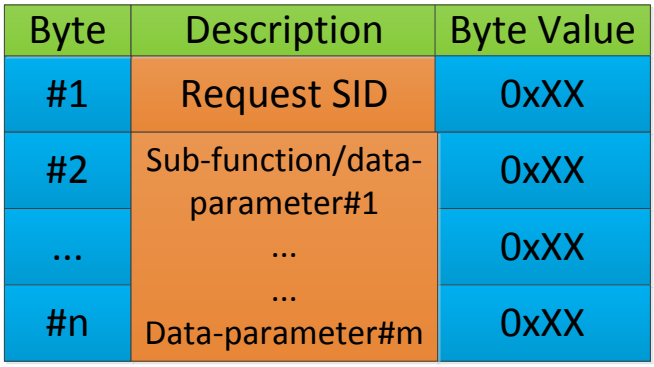

Figure 2-15: The structure of UDS-request message is defined in the UDS-standard, byte #1 is the request SID, byte #2 can be a sub-function of the UDS-service or the first data parameter. The rest of the bytes are data parameters.

The general format for a standard UDS-response is structured in a similar way to the request message as can be seen in Figure 2-16. A negative UDS-response will have a response SID that has byte value 0x7F on byte #1 as in Figure 2-17. Byte #2 will contain the request SID and a response code is included on byte #3.

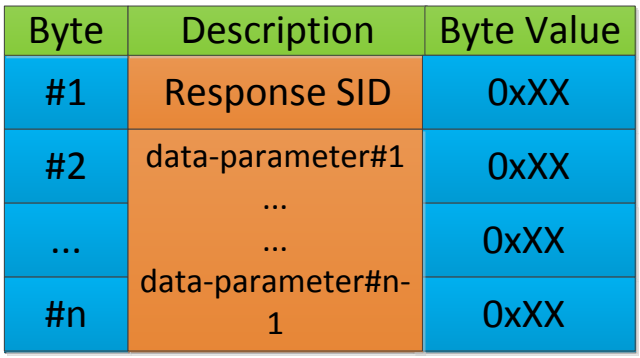

Figure 2-16: The structure of a UDS-response message is defined in the UDS-standard, byte #1 is the request SID, the rest of the bytes are data parameters.

Byte	Description	Byte Value
#1	Negative Response SID	0x7F
#2	Request SID	0xXX
#3	ResponseCode	0xXX

Figure 2-17: A negative UDS-response will have a response SID 0x7F, byte #2 will contain the request SID and byte #3 will contain a response code which contains information why the request was not accepted by the ECU.

A positive UDS-response will respond to a UDS-response with a positive response SID which is defined to be the request SID + 0x40 as in Figure 2-18. If the UDS-request contained a sub-function request then byte #2 will contain that byte value else it will contain the first data parameter, the rest of the bytes are data parameters.

Byte	Description	Byte Value
#1	Positive Response SID	0xXX+0x40
#2	Sub-function/data-parameter#1	0xXX
...	...	0xXX
#n	data-parameter#n-1	0xXX

Figure 2-18: A positive UDS-response will have a positive response SID which is the request SID + 0x40. Byte #2 can be sub-function of the UDS-service or the first data parameter. The rest of the bytes are data parameters.

Suppress positive UDS-response

There is a possibility for the diagnostic tester (client) to suppress positive responses from an UDS-request, which informs the ECU not to send a positive UDS-response. A suppress positive response request is done by setting bit #7 to 1 in the byte value representation of the sub-function. The bit value representation of a suppress positive response with sub-function 0x01 is illustrated in Figure 2-19.

Sub-function Byte Value = 0x81								
Bit #	7	6	5	4	3	2	1	0
Value	1	0	0	0	0	0	0	1

Figure 2-19: Bit value representation of a sub-function in an UDS-request. Bit #7 is set to 1 to indicate that it is a suppress positive response UDS-request.

The corresponding diagnostic session control request without suppress positive response can be seen in Figure 2-20.

Sub-function Byte Value = 0x01								
Bit #	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	0	0	1

Figure 2-20: Bit value representation the SID of a sub-function in an UDS-request.

Physical and functional addressing

There is a possibility for the diagnostic tester (client) to send physical or functional UDS-requests. A functional request is a broadcast-type message which will be sent to all ECU:s which are on the CAN-network. Physical UDS-requests are only sent to a single ECU on the network.

General negative response codes

When the ECU sends a negative UDS-response to the diagnostic tester (client) the response must include a response code on byte #3 as seen in Figure 2-17. There are some general negative response codes or NRC:s which are defined in the UDS-standard in addition to the UDS-service specific NRC:s. The NRC:s are divided into two different byte value ranges depending on the type of errors:

- 0x01-0x7F – Communication related NRC:s.
- 0x80-0xFF – NRC:s which are sent for specific conditions which are not correct at the time when the request is received by the server.

There are some general NRC:s related to issues that can occur during a software loading sequence. These are listed in Table 2-2. This project has mainly focused on the communication related NRC:s. But there are other tests which can be performed to generate NRC:s in the range 0x80 to 0xFF related to for example

incorrect vehicle speed, i.e. the car is moving during flashing of the software which is not a good idea.

Table 2-2: Communication related NRC:s commonly used in the UDS-services which are used in the software loading sequence

Byte Value	Negative Response Code (NRC) Definition
0x10	General Reject
0x11	Service Not Supported
0x12	Sub-function Not Supported
0x13	Incorrect Message Length Or Invalid Format
0x22	Conditions Not Correct
0x24	Request Sequence Error
0x31	Request Out Of Range
0x33	Security Access Denied
0x72	General Programming Failure
0x78	Request Correctly Received - Response Pending

- **General Reject (0x10):** Should only be used if none of the NRC:s defined in the UDS-standard meet the requirements of the implementation.
- **Service Not Supported (0x11):** Should be sent when the ECU receives a UDS-service request with an SID which is unknown or not supported.
- **Sub-function Not Supported (0x12):** Should be sent when the ECU receives a UDS-service request with an sub-function which is unknown or not supported.
- **Incorrect Message Length Or Invalid Format** (**0x13**): Should be sent when the length or format of the received request message does not match what is specified by service.
- **Conditions Not Correct** (**0x22**): Should be sent when the prerequisite conditions are not correct.
- **Request Sequence Error** (**0x24**): Should be sent when the ECU expected a different sequence of messages than what was sent by the tester.
- **Request Out Of Range** (**0x31**): Should be sent when the tester requests modifying a value which it does not have authority to change.
- **Security Access Denied** (**0x33**): Should be sent when the security requirements of the ECU are not fulfilled for the current request.
- **General Programming Failure** (**0x72**): Should be sent when the ECU has noticed an error while programming or erasing memory.

- **Request Correctly Received - Response Pending (0x78)**: Should be sent when the request is correctly formulated but the server has not yet completed the required actions. This NRC is supported in all UDS-services.

Session layer timings in the UDS-standard

Contained within the UDS-standard there is a standard governing the session layer services in the Open Systems Interconnection (OSI)-model called ISO14229-2. The most important session layer timings in this project are the P2 and P2 extended timings that specify the maximum time the server (ECU) or client (tester) has to wait or respond to an UDS-request. These values are communicated by the ECU through the UDS-response to the Diagnostic Session Control service, which is discussed in Chapter 2.8.3. P2 specifies the default timing which should be used, the ECU has however the option to send a NRC 0x78 (see Table 2-2) then the P2 extended timing value will be maximum time, which the server (ECU) has to respond.

2.8.3 UDS-services in the software loading sequences

Diagnostic Session Control

The UDS-service Diagnostic Session Control is used to control which diagnostic session the ECU should be in. There are a few different sessions, which are used for different purposes. The sessions, which are specified in the UDS-standard, are default-, programming-, extended- and safety system diagnostic session. On startup the ECU should be in the default session, then by using the diagnostic session control service the tester (client) may send UDS-requests to change the diagnostic session. An example of a diagnostic session control UDS-request can be seen in Figure 2-21.

Byte #	1	2	3	4	5	6	7	8
Value	0x10	0x02	-	-	-	-	-	-

Figure 2-21: A standard Diagnostic Session Control UDS-request to enter programming session. The request SID 0x10 on byte #1 and sub-function 0x02 on byte #2 according to the general message conventions.

The ECU will response to this message with either a positive or negative response if it is operational. A positive diagnostic session control service response will contain P2 and P2 extended timing values. These values represent the maximum time the ECU should take to return a response to a UDS-request in the

current session. The first value P2 is specified on bytes #3 and #4, P2 is the timing value used if the ECU has not sent a NRC 0x78 (see Table 2-2). P2 extended specified on byte #5 and #6 is the timing value in effect when the negative response code 0x78 has been sent by the ECU. The P2 and P2 extended values in Figure 2-22 are: P2 is equal to 10 ms and P2 extended is 500*10 ms = 5000 ms as the P2 extended value is specified in 10 ms resolution.

Byte #	1	2	3	4	5	6	7	8
Value	0x50	0x02	0x00	0x0A	0x01	0xF4	-	-

Figure 2-22: A standard positive Diagnostic Session Control UDS-response. It will return the response SID on byte #1 and the sub-function on byte #2 according to the general message conventions. In addition a positive Diagnostic Session Control service response will contain P2 and P2 extended timing values on bytes #3-6.

The negative response codes supported by the diagnostic session control are shown in Table 2-3.

Table 2-3: Negative response codes supported by the Diagnostic Session Control service

Byte Value	Negative Response Code (NRC) Definition
0x12	Sub-function Not Supported
0x13	Incorrect Message Length Or Invalid Format
0x22	Conditions Not Correct

ECU Reset service

The UDS-service ECU Reset is used to perform a reset of the ECU. This service is usually used in post-programming (see Chapter 2.8.4) after a reprogramming of an ECU. There are several sub-functions defined in the UDS-standard but only the sub-function “Hard Reset” is used in the software loading sequences used by BW-PDS customers (see Figure 2-23). After a ECU Reset has been performed the ECU should enter the default session.

Byte #	1	2	3	4	5	6	7	8
Value	0x11	0x01	-	-	-	-	-	-

Figure 2-23: A standard ECU Reset UDS-request with sub-function Hard Reset.

The negative response codes supported by the ECU Reset service are listed in Table 2-4.

Table 2-4: Negative response codes supported by the ECU Reset service

Byte Value	Negative Response Code (NRC) Definition
0x12	Sub-function Not Supported
0x13	Incorrect Message Length Or Invalid Format
0x22	Conditions Not Correct
0x33	Security Access Denied

Security Access service

In order to prevent unauthorized access to the ECU the vehicle manufacturers implement the Security Access service, which is specified in the UDS-standard. Generally security access is required before any transfer of new software to the ECU can be performed. The Security Access service utilizes a seed and key structure; the tester (client) will request security access with a UDS-request (see Figure 2-24).

Byte #	1	2	3	4	5	6	7	8
Value	0x27	0x01	-	-	-	-	-	-

Figure 2-24: Example of a request seed Security Access UDS-request.

The ECU will then respond with a positive response, which contains what is called a security seed, the length of the seed is vehicle manufacturer specific but is typically 3-4 bytes long (see Figure 2-25).

Byte #	1	2	3	4	5	6	7	8
Value	0x67	0x01	0xC6	0xF8	0x98	0x69	-	-

Figure 2-25: Example of a Security Access UDS-response containing the security seed.

The security seed will then be used in a vehicle manufacture specific algorithm, which will return a security, key that the tester (client) will send in an UDS-request (see Figure 2-26).

Byte #	1	2	3	4	5	6	7	8
Value	0x27	0x02	0xBF	0xFC	0xE7	0xC3	-	-

Figure 2-26: Example of a send key Security Access UDS-request containing the calculated security key.

If the security key returned by the tester (client) is correct the ECU will respond with a positive UDS-response (see Figure 2-27). After this the tester will be granted security access at the requested security level. The vehicle

manufacturers have the option of adding different security levels to differentiate what level of access is given to the user of the security access. Different levels of security access can be implemented by using different sub-function byte values to symbolize different levels of security. The relation between the request seed request and send key request is fixed so that if the request seed byte value is 0x01 then the send key byte value will be 0x02. If the request seed byte value is 0x03 then the send key byte value will be 0x04.

Byte #	1	2	3	4	5	6	7	8
Value	0x67	0x02	-	-	-	-	-	-

Figure 2-27: Example of a positive Security Access UDS-response granting security access to the tester (client).

The general negative response codes supported by the Security Access service are listed in Table 2-5 and the Security Access specific NRC:s are listed in Table 2-6.

Table 2-5: General negative response codes supported by the Security Access service

Byte Value	Negative Response Code (NRC) Definition
0x12	Sub-function Not Supported
0x13	Incorrect Message Length Or Invalid Format
0x22	Conditions Not Correct
0x24	Request Sequence Error
0x31	Request Out Of Range

Table 2-6: Negative response codes specific to Security Access service in the software loading sequence

Byte Value	Negative Response Code (NRC) Definition
0x35	Invalid Key
0x36	Exceeded Number Of Attempts
0x37	Required Time Delay Not Expired

- **Invalid Key (0x35):** Sent if the security key sent by the tester (client) does not match the expected key value.
- **Exceeded Number Of Attempts (0x36):** Sent if a delay timer is active due to too many incorrect send key Security Access requests.

- **Required Time Delay Not Expired (0x37):** Sent if the delay timer is still active.

Control DTC Setting service

The Control DTC Setting enables the tester to control the updating of the Diagnostic Trouble Codes (DTC:s) in the ECU:s. DTC:s are used to report possible faults present in the vehicle to for example a car repair shop and aid the debugging of problems experienced by the driver. For example a check engine light indicates that a DTC has been triggered. Generally the Control DTC Setting is used to disable the updating of DTC:s in the pre-programming step in a software loading sequence and then re-enabling the updating of DTC:s in post-programming (see Chapter 2.8.4). A typical Control DTC Setting UDS-request is shown in Figure 2-28.

Byte #	1	2	3	4	5	6	7	8
Value	0x85	0x01	0xFF	0xFF	0xFF	-	-	-

Figure 2-28: A typical Control DTC Setting UDS-request. This request contains an optional control option record on bytes #3-5 which determines which DTC-status bits should be affected by the request, in this case the request signals that all DTC-status bits should be enabled.

The negative response codes supported by the ECU Reset service are listed in Table 2-7.

Table 2-7: Negative response codes supported by the Control DTC Setting service

Byte Value	Negative Response Code (NRC) Definition
0x12	Sub-function Not Supported
0x13	Incorrect Message Length Or Invalid Format
0x22	Conditions Not Correct
0x31	Request Out Of Range

Communication Control service

This service is used by the tester to control transmission and/or reception of certain communication messages sent by the ECU:s. It will generally be used in conjunction with the Control DTC service in pre-programming and post-programming to disable/re-enable non-diagnostic communication. A typical Communication Control service UDS-request is shown in Figure 2-29.

Byte #	1	2	3	4	5	6	7	8
Value	0x28	0x01	0x01	-	-	-	-	-

Figure 2-29: A typical Communication Control service UDS-request, with sub-function EnableRxAndDisableTx (0x01) on byte #2 and communication type normalCommunication (0x01) on byte #3. Used to disable transmission of non-diagnostic communication.

Table 2-8: Negative response codes supported by the Communication Control service

Byte Value	Negative Response Code (NRC) Definition
0x12	Sub-function Not Supported
0x13	Incorrect Message Length Or Invalid Format
0x22	Conditions Not Correct
0x31	Request Out Of Range

Read By Identifier service

Service to read data at a memory location specified, used in a flashing sequence to read programming-, fingerprint-data and prepare the ECU for reprogramming. It is a vehicle manufacturer specific step that is sometimes included in the pre-programming part of the software loading sequence (see Chapter 2.8.4).

Byte #	1	2	3	4	5	6	7	8
Value	0x22	0xF1	0x58	-	-	-	-	-

Figure 2-30: A typical Read By Identifier UDS-service request. The data identifier of the memory location which should be read is specified on byte #2 and 3. The data identifier length is variable.

The general negative response codes supported by the Security Access service are listed in Table 2-9 and the Read By Identifier specific NRC is listed in Table 2-10.

Table 2-9: General supported negative response codes in the UDS-service Read By Identifier

Byte Value	Negative Response Code (NRC) Definition
0x12	Sub-function Not Supported
0x13	Incorrect Message Length Or Invalid Format
0x22	Conditions Not Correct
0x31	Request Out Of Range

Table 2-10: Read By Identifier specific negative response code in the software loading sequence

Byte Value	Negative Response Code (NRC) Definition
0x14	Response Too Long

- **Response Too Long (0x14):** Should be sent if the total length of the response exceeds the maximum length defined in the governing transport protocol.

Write By Identifier service

The Write By Identifier service is used to write data to a specific memory location, for example writing programming date and fingerprint data. Typically this is done right before transferring data to the ECU and/or after a successful software loading sequence. An example of a Write By Identifier UDS-request is shown in Figure 2-31.

Byte #	1	2	3	4	5	6	7	8
Value	0x2E	0xF1	0x58	0x15	0x11	0x19	0x02	0x03

Figure 2-31: A typical Write By Identifier UDS-service request. The data identifier of the memory location which should be written to is specified on byte #2 and 3. The following bytes represent the data record that should be written to that memory location. The data identifier and record length is variable.

The negative response codes supported by the ECU Reset service are listed in Table 2-11.

Table 2-11: Negative response codes supported by the Write By Identifier service

Byte Value	Negative Response Code (NRC) Definition
0x13	Incorrect Message Length Or Invalid Format
0x22	Conditions Not Correct
0x31	Request Out Of Range
0x33	Security Access Denied
0x72	General Programming Failure

Routine Control service

The Routine Control service is one of the most flexible services in the UDS-standard. It is typically used in the software loading sequence to check

programming pre-conditions and disable failsafe reactions in pre-programming (see Chapter 2.8.4). Also used for performing erasure of EEPROM or flash memory before downloading a new block, checking for valid flash memory and application after the transfer of data. The structure of a routine control service is shown in Figure 2-32.

One of the Routine Control services defined in the UDS-standard is the Erase Memory or Erase Flash routine. It will perform the erasure of EEPROM and flash memory before loading a new block. It has the routine identifier FF01, specified on bytes #3-4 in the UDS-request. The remaining bytes of the request contain a routine control option record which is of variable length depending on vehicle manufacturing specification and which routine identifier is used. An example of a start Erase Flash Routine Control service is shown in Figure 2-33.

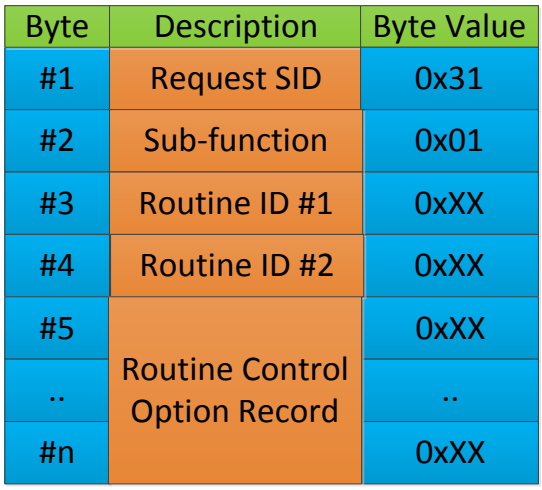

Figure 2-32: Start Routine Control Service UDS-request. Sub-function byte value 0x01 indicates Start Routine. The routine identifier (ID) is written on bytes #3-4.

Byte #	1	2	3	4	5	6	7	8
Value	0x31	0x01	0xFF	0x00	0x01	0x15	-	-

Figure 2-33: A typical Erase Flash Routine Control UDS-request. This request aims to start an erasure of a EEPROM or flash memory sector before downloading block 0x15.

The negative response codes supported by the Routine Control service are listed in Table 2-12.

Table 2-12: Negative response codes supported by the Routine Control service

Byte Value	Negative Response Code (NRC) Definition
0x12	Sub-function Not Supported
0x13	Incorrect Message Length Or Invalid Format
0x22	Conditions Not Correct
0x24	Request Sequence Error
0x31	Request Out Of Range
0x33	Security Access Denied
0x72	General Programming Failure

Request Download service

The Request Download service together with Transfer Data and Request Transfer Exit constitute the main components of the actual transfer of new block data to a reprogrammable ECU. A Request Download UDS-request contains information of which memory address the data block should be downloaded to, how large the data block is in bytes and if the data block is encrypted and/or compressed. The structure of a Request Download UDS-request can be seen in Figure 2-34.

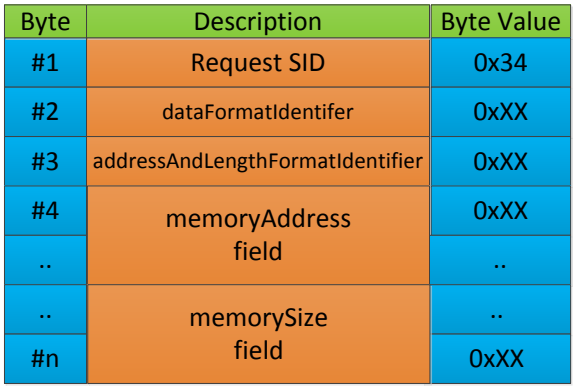

Figure 2-34: Structure of a Request Download Service UDS-request. The data format identifier specifies if data is encrypted and/or compressed. The address and length format identifier specifies the size of the memory address and size fields in bytes. The memory address indicates where the data should be written and the memory size field specifies how many bytes are in the block to be downloaded.

The data format identifier specifies if data is encrypted and/or compressed. The address and length format identifier specifies the size of the memory address and size fields in bytes. The memory address indicates where the data should be

written and memory size specifies how many bytes are in the block to be downloaded.

Examples to illustrate how the address and length format identifier affects the size of the memory address and size fields can be seen in Figure 2-35 and Figure 2-36. Some projects use a start address as the memory address while other projects use a block number to indicate the memory address.

Byte #	1	2	3	4	5	6	7	8
Value	0x34	0x00	0x41	0x31	0x00	0x00	0x05	0x34

Figure 2-35: An example of a Request Download UDS-request with address and length identifier on byte #3 with a byte value 0x41. The 4 represented by bits #7-4 on byte #3 indicate that the memory size is written on 4 bytes. The 1 represented by bits #3-0 on byte #3 indicates that the memory address field is written on 1 byte. Bytes #5-8 indicate how large data block should be downloaded, in this case 1332 bytes (0x0534).

Byte #	1	2	3	4	5	6	7	8	9	10	11
Value	0x34	0x00	0x44	0x00	0xF1	0x00	0x00	0x00	0x00	0x05	0x34

Figure 2-36: An example of a Request Download UDS-request with address and length identifier on byte #3 with a byte value 0x44. The 4 represented by bits #7-4 on byte #3 indicate that the memory size is written on 4 bytes. The 4 represented by bits #3-0 on byte #3 indicate that the memory address field is written on 1 byte. Bytes #8-11 indicate how large data block should be downloaded, in this case 1332 bytes (0x0534).

The response from a Request Download UDS-request contains information about how large Transfer Data requests are accepted by the ECU. The UDS-standard requires that the ECU must be able to accept at least a Transfer Data request of the length specified by the "MaxNumberOfBlockLength" contained in the response. Transfer Data requests of shorter length than what is indicated by the "MaxNumberOfBlockLength" are vehicle manufacturer specific if they accept or not. However, the last Transfer Data message may contain less than the maximum size of data packages, which must be handled by the ECU according to the UDS-standard.

Byte #	1	2	3	4	5	6	7	8
Value	0x74	0x20	0x0C	0x62	-	-	-	-

Figure 2-37: An example of a Request Download UDS-response. On byte #2 the length format identifier indicates on how many bytes the "MaxNumberOfBlockLength" should occupy. The 2 represented by bits #7-4 on byte #2 indicates that the "MaxNumberOfBlockLength" will be written on two bytes. On byte #3-4 the "Max NumberOfBlockLength" is equal to 3170 (0x0C62) in this case. This value includes the block sequence counter and request SID of the Transfer Data Service.

The negative response codes supported by the Request Download service are listed in Table 2-13.

Table 2-13: The negative response codes supported by the Request Download service

Byte Value	Negative Response Code (NRC) Definition
0x12	Sub-function Not Supported
0x13	Incorrect Message Length Or Invalid Format
0x22	Conditions Not Correct
0x24	Request Sequence Error
0x31	Request Out Of Range
0x33	Security Access Denied
0x72	General Programming Failure

- **Request Out Of Range (0x31) –** Should be sent out if any of the data fields in Figure 2-34 are not valid, i.e. dataformatIdentifier, addressAndLengthFormatIdentifier, memory address and size fields are invalid (see Figure 2-34).

Transfer Data service

The data in the software loading sequence is sent out in Transfer Data service-requests. After the tester has received a positive Request Download response the client (tester) will be permitted to send Transfer Data requests.

The Transfer Data request contains a block sequence counter, which can be used in error handling during the data transfer. The block sequence counter should start at 0x01, and then add 1 (one) every new Transfer Data request and when it reaches 0xFF it should roll over to 0x00. If two Transfer Data requests received by the ECU contain the same block sequence counter the ECU should send positive responses on both. It is recommended that the ECU only write the data to the flash memory from the first Transfer Data request with the same block sequence counter. A typical Transfer Data request sequence is shown in Figure 2-38. This example shows the transfer of a block of size 1076 bytes. This ECU has a "MaxNumberBlockLength" of 514 bytes including the request SID and block sequence counter. This means that 512 bytes will be sent in the first two Transfer Data requests and the last request will contain the last 52 bytes in the block file.

Byte #	1	2	3	Data bytes..			MaxNumberOfBlockLength
Value	0x36	0x01	0xXX	0xXX	.....	0xXX	0xXX

Byte #	1	2	3	Data bytes..			MaxNumberOfBlockLength
Value	0x36	0x02	0xXX	0xXX	.....	0xXX	0xXX

Byte #	1	2	3	Data bytes..			54
Value	0x36	0x03	0xXX	0xXX	.....	0xXX	0xXX

Figure 2-38: A typical Transfer Data request sequence. This example shows the transfer of 1076 bytes. The ECU has a "MaxNumberBlockLength" of 514 bytes including the request SID and block sequence counter. This means that 512 bytes will be sent in the first two Transfer Data requests and the last request will contain the last 52 bytes in order to complete the sending of 1076 data bytes.

The negative response codes supported by the Transfer Data service are listed in Table 2-14 and the specific NRC:s for the Transfer Data service in the software loading sequence are listed in Table 2-15.

Table 2-14: General supported negative response codes in the UDS-service Transfer Data

Byte Value	Negative Response Code (NRC) Definition
0x13	Incorrect Message Length Or Invalid Format
0x24	Request Sequence Error
0x31	Request Out Of Range
0x72	General Programming Failure

Table 2-15: Transfer Data specific negative response codes in the software loading sequence

Byte Value	Negative Response Code (NRC) Definition
0x71	Transfer Data Suspended
0x73	Wrong Block Sequence Counter
0x92/0x93	Voltage Too High / Voltage Too Low

- **Transfer Data Suspended (0x71):** If the memorySize (see Figure 2-34) parameter in the Request Download request does not match the number of bytes sent by the Transfer Data requests this NRC should be sent by the ECU.
- **Wrong Block Sequence Counter (0x73):** If the ECU notices an error in the "blockSequenceCounter" sequence then it should send this NRC.

- **Voltage Too High / Voltage Too Low (0x92/0x93):** Should be sent if the voltage measured by the ECU are outside of the permitted range for a software loading sequence.

Request Transfer Exit service

The Request Transfer Exit UDS-request should be sent after all the data has been transferred with the Transfer Data requests. A typical Request Transfer Exit UDS-request in the software loading is shown in Figure 2-39.

Byte #	1	2	3	4	5	6	7	8
Value	0x37	-	-	-	-	-	-	-

Figure 2-39: A typical Request Transfer Exit UDS-request in the software loading.

The format of the Request Transfer Exit response is vehicle manufacture specific; some of BW-PDS customers will send the checksum for the downloaded block in the response (see Figure 2-40). In these cases it is the responsibility of the tester (client) to check that this checksum matches and if it does not the tester should abort the software loading sequence to avoid loading the incorrect data onto the ECU. Other manufacturers use a separate Routine Control in which the tester sends the checksum to the ECU and the ECU will handle the possible checksum error instead.

Byte #	1	2	3	4	5	6	7	8
Value	0x77	0xFF	0x7A	-	-	-	-	-

Figure 2-40: Request Transfer Exit response with a block checksum included in the response.

The negative response codes supported by the Request Transfer Exit are listed in Table 2-16.

Table 2-16: Negative response codes supported by the Request Transfer Exit service

Byte Value	Negative Response Code (NRC) Definition
0x13	Incorrect Message Length Or Invalid Format
0x24	Request Sequence Error
0x31	Request Out Of Range
0x72	General Programming Failure

Tester Present service

The Tester Present UDS-service request is sent by the tester (client) to keep the ECU in the current diagnostic session, which will prevent the ECU from returning to the default diagnostic session, which would otherwise happen after a predetermined time.

The negative response codes supported by the Tester Present service are listed in Table 2-17.

Table 2-17: Negative response codes supported by the Tester Present service

Byte Value	Negative Response Code (NRC) Definition
0x12	Sub-function Not Supported
0x13	Incorrect Message Length Or Invalid Format

2.8.4 Standardized software loading sequence

Within the UDS-standard ISO14229-1 there is a framework defined for *"non-volatile server memory programming process* [19, p. 303]*"* or software loading sequence for reprogrammable ECU:s in other words. It contains specifications for the entire software loading sequence with mandatory; optional/recommended and vehicle manufacture specific steps to accommodate different vehicle manufacturers preferences but at the same time keep the flashing sequence relatively similar regardless of which vehicle manufacturer specific flashing sequence is used.

The software loading sequence defined in the UDS-standard is divided into two main programming phases. Programming phase #1 – download of application software and/or application data and programming phase #2 – server configuration that is an optional phase. These programming phases are in turn divided into three sub-steps: pre-programming, programming and post-programming (see Figure 2-41).

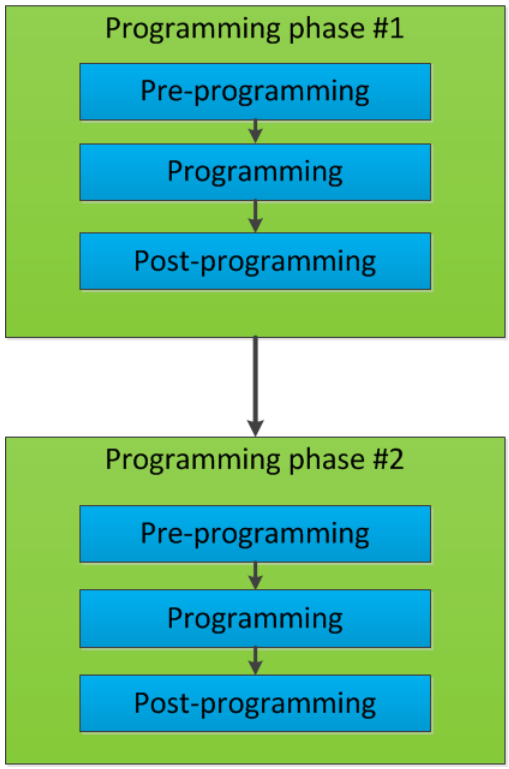

Figure 2-41: Non-volatile server memory programming process framework defined in the UDS-standard ISO14229-1

Pre-programming in phase #1

The pre-programming step in programming phase #1 is an optional step, which configures the ECU before the actual transfer of data. The mandatory steps included in this pre-programming step are sending a Diagnostic Session Control (0x10) UDS-request in order to enter the extended diagnostic session (0x03) on the ECU. When the ECU is in extended session it is possible for the tester to send UDS-requests Control DTC Setting (0x85) and Communication Control (0x28) to disable updating of DTC:s and disable non-diagnostic communication. Other optional steps include checking programming preconditions and disable fail-safe reactions with a Routine Control (0x31) UDS-request. Some manufacturers also use Read Data By Identifier (0x22) to read ECU data.

Programming in phase #1

The actual downloading of application software and/or application data is done in the programming step in programming phase #1. The mandatory steps included in

the programming phase are to enter programming session (0x02) by using a Diagnostic Session Control (0x10) UDS-request. Then sending the main downloading sequence UDS-requests Request Download (0x34), Transfer Data (0x36) and Request Transfer Exit (0x37). There are also several optional/recommended steps such as Security Access (0x27) with the seed and key scheme, Erase Memory (0xFF00) and various checks on EEPROM or flash memory by using the Routine Control (0x31) UDS-service.

Post-programming in phase #1

The post-programming step of programming phase #1 only contains one step which can either be a ECU Reset (0x11) or a Diagnostic Session Control (0x10) request to put the ECU in the default diagnostic session.

Pre-programming in phase #2

The pre-programming step in phase #2 – server configuration is performed by sending a request to enter the extended diagnostic session to for example enable the control of updating of DTC:s and re-enable communication with the Control DTC Setting (0x85) and Communication Control (0x27) UDS-services.

Programming in phase #2

In the programming step in phase #2 it is mandatory to clear diagnostic information, which might have been stored in the re-programmable ECU with a Clear Diagnostic Information (0x14) UDS-service. This step was, however, not performed by any of the projects at BW-PDS, which were analyzed in this project. There is an option to include vehicle manufacturer specific options in this step.

Post-programming in phase #2

The post-programming step of programming phase #2 only contains one step which can either be a ECU Reset (0x11) or a Diagnostic Session Control request to put the ECU in the default diagnostic session just like the post-programming step in phase #1.

2.9 Diagnostic communication over CAN ISO15765-2

The Diagnostic communication over Controller Area Network (DoCAN) for road vehicles or ISO15765-2 is the governing standard for transport protocol layer (OSI layer 4) and network layer services (OSI layer 3) for the implementation used in

this project. Other standards for implementation of the transport and network layers in the OSI-model for the UDS-standard include Communication on FlexRay (ISO10681-2), Diagnostic communication over Internet Protocol (DoIP) for road vehicles (ISO13400-2) and Local Interconnect Network (LIN) for road vehicles (ISO17987-2) [19, p. vii].

The international standard ISO15765-2 contains specifications of how for example a data frame, which cannot fit into a single CAN-frame, should be handled by using segmented messages and the minimum time between consecutive frames allowed SeparationTime minimum (STmin).

Most of the transport and network specific issues are handled automatically by CANoe through the diagnostic communication interface (see Chapter 4.2). Therefore the details contained within the ISO15765-2 have not been of such importance to the implementation of this project, which has mostly dealt with issues regarding the session and application layers (layers 5 and 7) in the OSI-model [20].

2.10 AUTOSAR

The AUTomotive Open System ARchitecture (AUTOSAR) is a partnership between several vehicle manufactures and automotive suppliers established in order to create an open industry standard for automotive ECU:s. The goal is to create an architecture that promotes the reuse of software, enable collaboration between several different manufacturers and provide scalable systems, which are "Commercial of the Shelf" solutions more frequently.

According to a Master's thesis from Jönköping University there are two protocols in the AUTOSAR-standard, which specify how to implement the software loading sequence. These are UDS and Universal Measurement and Calibration Protocol (XCP) specified by Association for Standardization of Automation and Measuring Systems (ASAM). This project relies heavily on the UDS-standard and according to the authors of the Master's thesis from Jönköping University the UDS-standard is the primary protocol used to implement the "*AUTOSAR specification of Flash Driver"* [21] [22].

3. Equipment

The equipment used in this project was ECU:s from several different customers similar to the one depicted in Figure 3-2. It has two main connectors; the larger connector is used for the CAN-communication and the smaller one is connected to a pump that regulates how the torque is applied by the coupling. Several of BW-PDS customers let the software-testing department in Landskrona perform tests on the ECU-software that controls the coupling in the torque transfer systems.

The microcontrollers that are used in modern ECU:s generally have relativity limited hardware in order to keep the cost down of the final vehicle. ECU:s typically have a few kilobytes of a volatile memory type like SRAM integrated on the microcontroller and a few megabytes of non-volatile memory such as EEPROM or flash memory to store the software on the ECU.

During the software loading sequence or flashing in this project the ECU is connected to a real-time system VN8970 (see Figure 3-1) from Vector Informatik GmbH.

Figure 3-1: VN8970 real-time system from Vector Informatik GmbH for simulating the CAN-network in which the ECU will later be used. Figure by A. Karlsson [23].

The VN8970 system provides the possibility of simulating how the ECU will perform in the conditions in which it will later be used in the vehicle. It has 4 channels that can be used for CAN or LIN communication as well as a configurable digital/analog channel. An ATMEL AT91SAM9 processor powers the real-time system in the VN8970.

A CAN/LIN to USB interface module VN1610 from Vector Informatik GmbH was used to connect the PC to the simulated CAN-network. The ECU is connected to a power supply, which can be controlled from CANoe.

For software development Microsoft Visual Studio 2013 was used for the binary file converter (see Chapter 4.1) and the rest of the development for CANoe was done in the Vector CAPL Browser.

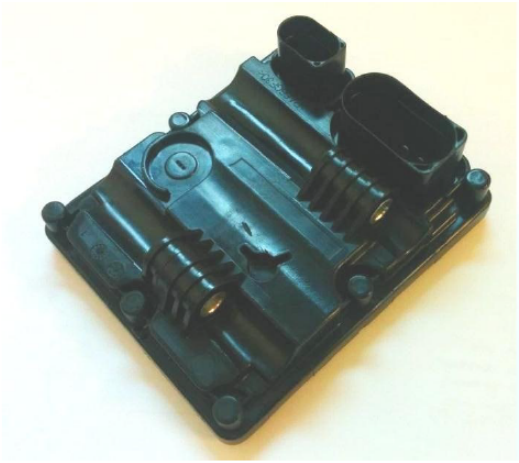

Figure 3-2: Electrical Control Unit used in a BW-PDS coupling in a torque transfer system. Figure by A. Karlsson [23].

4. Implementation

In order to get the software loading process into BW-PDS´s test environment CANoe, vendor-specific data has to be converted into the CIN-file format (see Figure 4-1) that can be read by CANoe. The conversion of vendor-specific data is done by a converter application, which was developed in this project and is presented in Chapter 4.1. The data extracted is then used for the software loading process in CANoe.

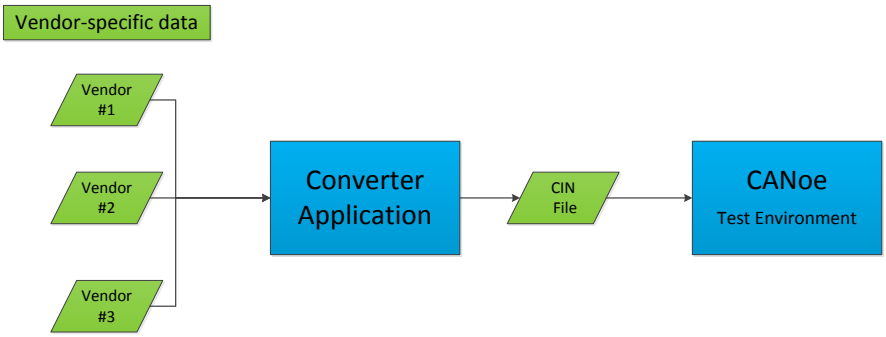

Figure 4-1: Structure of solution for the binary data conversion. The vendor-specific data is converted into a CIN-file by a converter application, which can be read by CANoe.

The software loading process is performed in CANoe by using the structure of the CAPL-solution presented in Figure 4-2. The common UDS flash sequence discussed in Chapter 4.4, which was developed in this project contains the vehicle manufacturer specific software loading sequences. It also contains commonly used test cases on the software loading sequence. These test cases can be called from a project specific CAN-file (see Chapter 2.6.1). This structure enables the user to choose which test cases should be performed on the software loading sequence in a software test and add project specific flash test cases in a separate file.

The flashing sequence is then performed by utilizing the functions in common UDS flash services (see Chapter 4.3) to perform the actual software

loading sequence. Common UDS flash services, which was also developed in this project, contain all the functions that are commonly used in the UDS-software loading sequence. The diagnostic communication interface (see Chapter 4.2) performs the actual sending of the UDS-messages in on the CAN-bus.

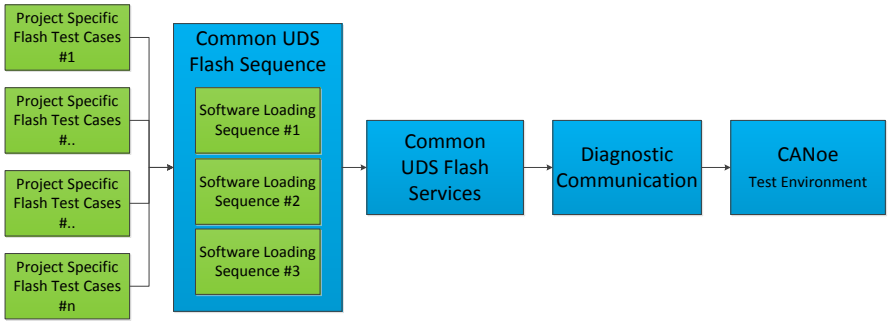

Figure 4-2: Structure of CAPL-solution, which performs the software loading process according to each vehicle manufacturers specification. Common UDS flash sequence contains the commonly used test cases, which can be used in different project specific, flash test cases. The project specific flash test cases determine which test cases to be performed. The Common UDS flash services interface contains functions to send the UDS-services used in the flashing sequence. The diagnostic communication interface utilizes the UDS-support in CANoe to send messages to the ECU.

4.1 Converter application

A C# Windows Forms application was developed in this project to convert the vendor-specific binary data format in to a .CIN file, which can be read by CANoe. The structure of the converter application can be seen in Figure 4-3. The file parsers used in the internally developed flashing tool were integrated into the converter application. The graphical user interfaces created for the converter application also took inspiration from the current flashing tool, as the requirements are similar. Some modifications were however made to simplify the application, as this application will be used by a more limited user group than the internally developed flashing tool used in production. There is also no need of a "save configuration" option as once the binary files have been converted the files needed for the software loading process in that project are available to use. Therefore in theory the conversion of the binary files only has to be done once every new software release.

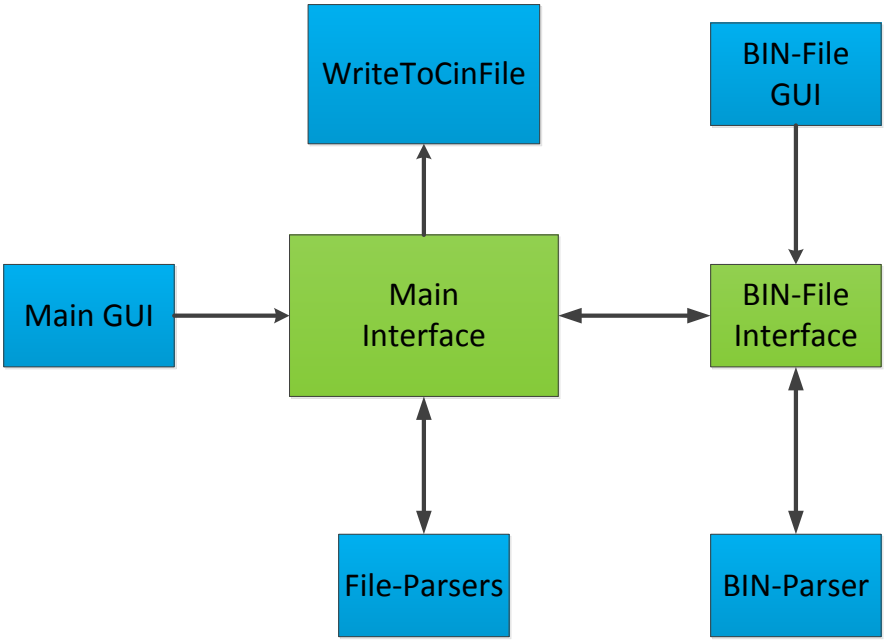

Figure 4-3: Structure of the converter application. A special interface was needed for the BIN-file format. Arrows indicate the data exchange in the application.

4.1.1 Main graphical user interface and BIN-file interface

The main graphical user interface (GUI) is used for most file formats that contain all the data needed for a software loading sequence. The main GUI can be seen in Figure 4-4. The BIN-file format does not contain an identifier so it is not possible to separate the file blocks from each other based on the data contained in the file. Therefore a separate interface was created in which the user adds the binary data files to their corresponding file block identifier. The separate graphical user interface is depicted in Figure 4-5. This GUI is similar to the GUI in the internally developed flashing tool (see Figure 2-8). The user can mark a selected row and then press the button "Add Bin File To Selected Row" and the interface will then automatically shift down to the next row. When all rows are filled the user can press the convert button that will output a .CIN-file in the same way as for the other file formats.

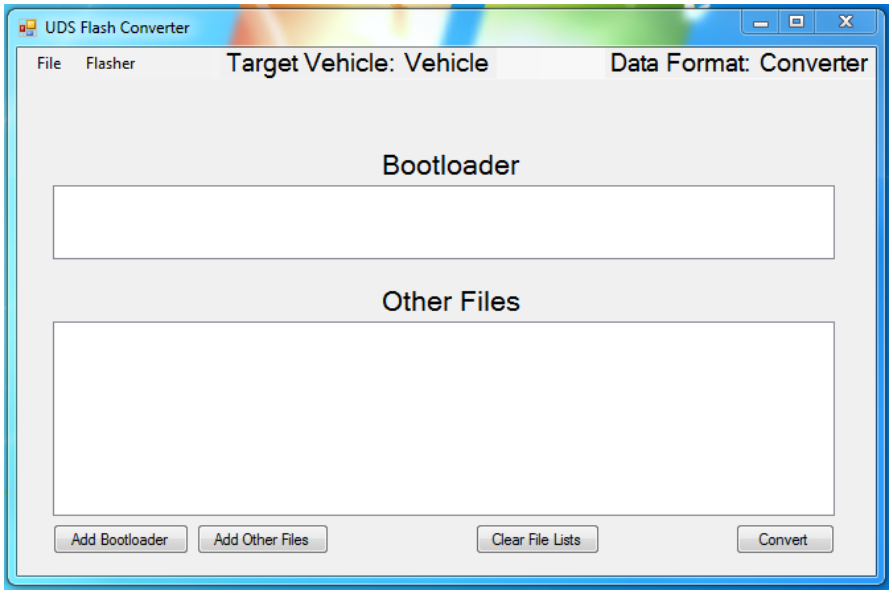

Figure 4-4: The main graphical user interface used for most file formats. First the user selects the target flasher, and then adds the secondary bootloader, then the other files. The application will then output the CIN-files to a folder called FlashingFiles when the convert button is pressed.

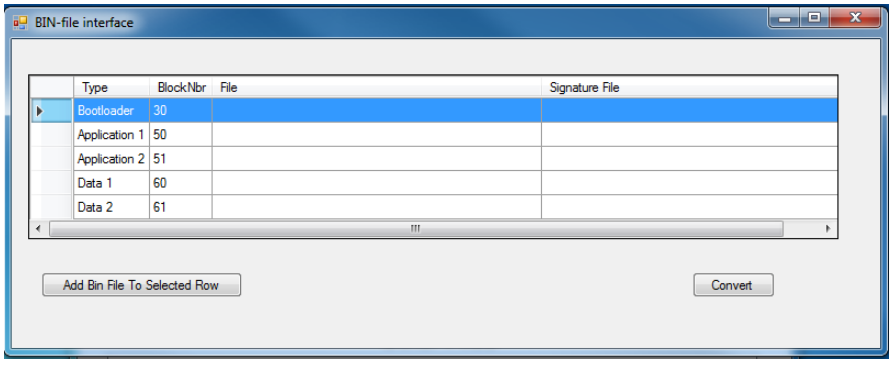

Figure 4-5: Special interface for the BIN-file format. The user adds the files to the corresponding block number (or the block identifier). A special signature file is loaded automatically if it is in the same folder with the correct filename.

4.1.2 Generic data format for the flashing sequence

In this project a generic binary data file format was developed in order to enable implementation of common test cases on the software loading process specified by the UDS-standard. Several of BW-PDS customers use different binary data

formats for containing the data needed for the software loading sequence (see Chapter 2.2 for more details on this). This created a challenge to develop a generic format that would work for all the manufacturers. The different binary data formats contained varying amount of information needed for the vendor-specific flashing sequences and also had different numbers of binary data files. It was however possible to create a generic format for BW-PDS's customers. The generic file-format developed in this project is divided in two different types of CIN-files which are depicted in Figure 4-6 and Figure 4-7.

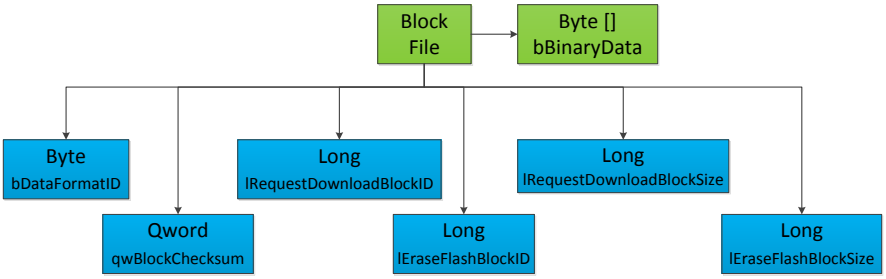

Figure 4-6: The generic format for the block binary data-files used in the flashing sequence in this project.

The generic format for block binary data-file contains a header with variables, which are used in the flashing sequence to perform download of the binary data included in the block files. These variables are explained in the list below.

- **bBinaryData** – The binary data to be transferred in the software loading sequence.
- **bDataFormatID** – Is used in the dataFormatIdentifer field in the Request Download service to specify if the block is compressed and/or encrypted.
- **qwBlockChecksum** – Contains the checksum for the binary data block which is being downloaded.
- **lRequestDownloadBlockID** – Is used in the memoryAddress field in the Request Download service as an identifier for the binary data block.
- **lRequestDownloadBlockSize** – Is used in the memorySize field in the Request Download service to specify the size of the block to be downloaded in the Transfer Data request.
- **lEraseFlashBlockID** – Similar function as the RequestDownloadID variable but used in the Erase Flash Routine Control UDS-request.

- **lEraseFlashBlockSize** - Similar function as the RequestDownloadBlockSize variable but used in the Erase Flash Routine Control UDS-request.

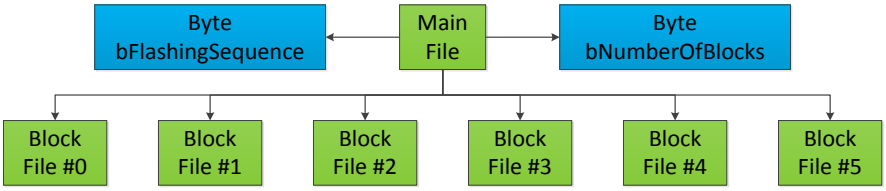

Figure 4-7: The generic format for the main file which includes the block binary data-files (see Figure 4-6) for the flashing sequence in this project.

The main file, which includes all of the binary data block files for the flashing sequence, also contains two variables which are used to control the software loading sequence according to the corresponding project specification. These are described in the list below.

- **bNumberOfBlocks** – Number of real blocks contained in the projects. This number excludes the dummy blocks which are added to avoid compilation errors in the CAPL-code.
- **bFlashingSequence** – Used to control which vendor specific flashing sequence (see Figure 4-2) is run during the flash tests.

4.2 Diagnostic communication interface

At the software testing department at BW-PDS a diagnostic communication interface has been developed using the diagnostic CAPL functions (see Chapter 2.6). This simplifies the process of sending the Unified Diagnostic Service (UDS) messages further. An example of a function sending a simple UDS-message using the interface is shown in Figure 4-8.

```
void ControlDTCSetting(byte bDTC_Setting)
{
/*****************************************************************************
* FUNCTION NAME ControlDTCSetting
* DESCRIPTION   Sends out message to ECU to disable or enable DTC-MESSAGES.
* PARAMETERS    bDTC_Setting:   bDTC_TURN_ON or bDTC_TURN_OFF
* RETURN VALUE  None.
*****************************************************************************/
    byte bRequest[5];

    bRequest[0] = 0x85;
    bRequest[1] = bDTC_Setting;

    iDCSendRequest(bRequest, 2);

    iDCTestWaitForDiagResponse(5000);

    //Reset communication type to physical
    DCSetCommunicationType(iPHYSICAL);

}
```

Figure 4-8: Code for the function to send a simple UDS-message. This code sends a ControlDTCSetting message (see Table 2-1) to the specified ECU. This is a function contained in the common UDS flash services interface.

There are also helpful features when reading UDS-responses from the ECU. An example of the code for reading a UDS-response is shown in Figure 4-9. The function iDCGetRespPrimitiveSize() gets the size of the last received UDS-message, then a simple code to extract the maximum block size allowed according to the RequestDownload UDS-response (see Table 2-1). The function iDCGetRespPrimitiveByte() returns the byte at the position specified by the input parameter.

```
byte bInResponse[2];
byte bMaxNumberOfBlockLengthByteArray[2];

iInPrimativeSize = iDCGetRespPrimitiveSize();

for(i=0; i<iInPrimativeSize-2; i++){
    bInResponse[i] = iDCGetRespPrimitiveByte(i+2);
}
bMaxNumberOfBlockLengthByteArray[0] = bInResponse[0];
bMaxNumberOfBlockLengthByteArray[1] = bInResponse[1];
```

Figure 4-9: Example of reading an UDS-message with the diagnostic communication interface. This example reads the maximum download block size allowed by the ECU

from the RequestDownload UDS-response (see Figure 2-37). The first two bytes in the response contain the maximum size permitted.

4.2.1 Modifications to diagnostic communication interface

A few modifications and additions have been made to the existing diagnostic communication interface during the project to support a complete software downloading process. These modifications mainly regard the size of the UDS-messages. Some of the CAN-data frames used in the software loading process were larger than what previously had been tested when using the diagnostic communication interface. Also some new features were added to the diagnostic communication interface: enable or disable suppression of positive response UDS-messages and a function to check if the last response received was positive.

4.3 Common UDS flash services

The common UDS flash services interface contains all the methods for sending the different UDS-service messages needed for the flashing sequence using the diagnostic communication interface (see Chapter 4.2).

Each UDS-service used in the software loading sequence (see Table 2-1) has a corresponding function in the common UDS flash services interface. One example of the functions in the interface is the ControlDTCSetting function in Figure 4-8, which represents UDS-service with the SID 0x85. The required DTC-setting is sent as a parameter to this function. Another more advanced example of a UDS-service is the diagnostic session control. The function used for setting the correct session in the common UDS flash services interface can be seen in Figure 4-10. In addition to sending the UDS-request the function will update the timing values P2 and P2 extended (see Chapter 2.8.2) to represent the current session values. The function will also write the default diagnostic session timing values, which will be used in the ECU Reset function.

```
void SetDiagnosticSessionControl(byte bInDiagnosticSession)
{
/*****************************************************************************
* FUNCTION NAME SetDiagnosticSessionControl
* DESCRIPTION   Sends DiagnosticSessionControl UDS-message to ECU, Sets P2/P2ex
*               according to diagnostic session response
* PARAMETERS    bInDiagnosticSession - Specifice session.
*               Ex: bDEFAULT_DIAGNOSTIC_SESSION, bEXTENDED_DIAGNOSTIC_SESSION
*                   bPROGRAMMING_DIAGNOSTIC_SESSION
* RETURN VALUE  None
*****************************************************************************/
    byte bRequest[2];

    bRequest[0] = 0x10;
    bRequest[1] = bInDiagnosticSession;

    iDCSendRequest(bRequest, 2);
    iDCTestWaitForDiagResponse(5000);

    //Help function to set P2 and P2star, Only if positive response
    if(iDCCheckIfPositiveResponse() != iFALSE)
    {
        HelpFunctionSetP2_P2starFromValuesInDiagnosticSessionControl();
        if(bDEFAULT_DIAGNOSTIC_SESSION == bInDiagnosticSession)
        {
            HelpFunctionWriteP2ValuesFromDefaultSession();
        }
    }
    //Reset communication type to physical
    DCSetCommunicationType(iPHYSICAL);
}
```

Figure 4-10: Set Diagnostic Session Control function in the common UDS flash services interface. It will send UDS-service message to initiate a diagnostic session change on the ECU. The parameter to the function determines which session is requested. After a positive UDS-response has been received from the ECU the function will set the P2/P2 extended timing values in CANoe according to the response. The function will also write the default diagnostic session timing values, which will be used in the ECU Reset function.

4.4 Common UDS flash sequence

The common UDS flash sequence interface contains the software downloading sequence for the customer's projects at the software-testing department at BW-PDS. By using markers/flags the flashing sequence is performed according to the respective vehicle manufacturer specification. The common UDS flash sequence interface also contains the commonly used test cases, which make it possible to run several different flash tests based on the UDS-standard by simply calling the commonly used test cases with a parameter corresponding to the requested flash

test case. The parameter corresponding to the specific flash test will then by using markers/flags perform the flash test by modifying the main downloading sequence UDS-services Request Download (0x34), Transfer Data (0x36) and Request Transfer Exit (0x37) (see programming step in programming phase #1 in Chapter 2.8.4).

The flash sequences contained in the common UDS flash sequence interface are divided into a few sub-steps which differ slightly from the sub-steps presented in the "non-volatile server memory programming" in the UDS-standard. The reasons for the differences in the grouping are due to the effort making the flashing sequence more modular and graspable. This is important when automatically generating test reports, running automated software test cases on the flashing sequence and also adding the possibility to change the download sequence and thereby making it possible to easy to create new commonly used test cases.

4.4.1 Commonly used test cases

Each sub-step in the flashing sequence is contained within a CAPL-test function. These test functions can then be combined in order to create commonly used test cases.

Download with errors on all blocks except SBL

One example of a commonly used test case is "Download with errors on all blocks except SBL" (see Figure 4-11) in which it is possible add parameters that will alter the flashing sequence with the corresponding modification for the flash test case.

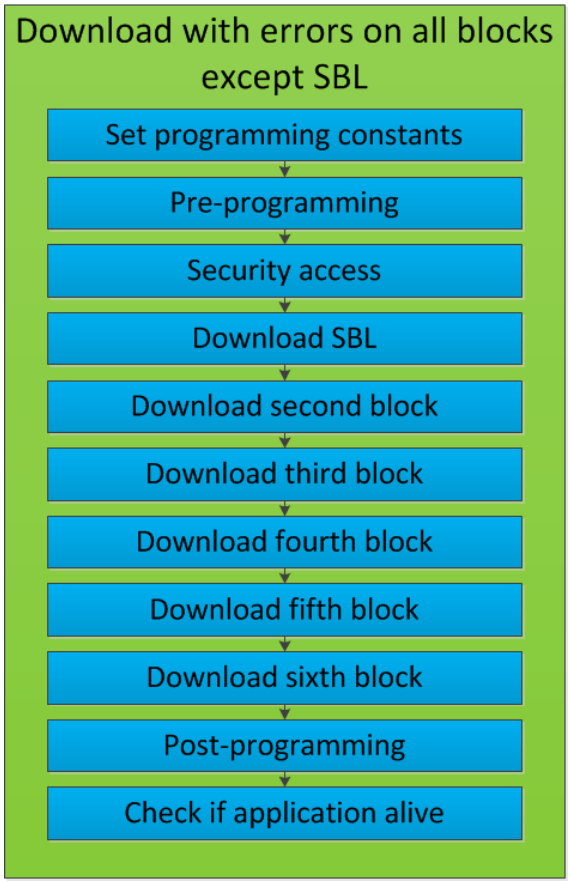

Figure 4-11: Flow chart for the most commonly used test case "Download with errors on all blocks except SBL".

The commonly used test case "Download with errors on all blocks except SBL" is the most utilized test case in this project. The reasons for this is that it performs the flashing sequence in the normal way with errors on all blocks except for the Secondary Boot Loader (SBL). If the errors had also been injected on the SBL then it would severely affect the downloading process of the following blocks as it is the SBL that performs the downloading of the other blocks (see Chapter 2.7). The function of each test function in Figure 4-11 can be seen in the list below.

- **Set programming constants:** Sets various constants according to the specific vehicle manufacturers specification, such as security access type, addressAndLengthIdentifier and default max block length allowed in Transfer Data (0x36, see Chapter 2.8.3).

- **Pre-programming –** Performs the pre-programming according to the specification of the pre-programming in programming phase #1 in the framework specified by the UDS-standard (see Chapter 2.8.4).
- **Security Access –** Performs the seed and key sequence (see Chapter 2.8.3), which is included in the programming step in programming phase #1 in the UDS-specification (see Chapter 2.8.4).
- **Download SBL –** Performs download of the SBL with UDS-services Request Download 0x34, Transfer Data 0x36 and Request Transfer Exit (0x37). Some vehicle manufacturers perform an activation of the SBL after downloading it. Erase Flash (0xFF) Routine Control is not performed on the SBL as the SBL is loaded onto the volatile memory type RAM, which means that it will not be available after a ECU reset. Otherwise the downloading of SBL is identical to the process for the flowing blocks.
- **Download second to sixth block -** Performs download of corresponding block with UDS-services Request Download (0x34), Transfer Data (0x36) and Request Transfer Exit (0x37). The number of blocks varies from manufacturer to manufacturer; therefore less than six blocks may be downloaded.
- **Post-programming –** Performs various Routine Controls (0x31) to verify a correct flashing sequence, resets communication and DTC related options on the ECU by using Communication Control (0x28) and ControlDTCSetting (0x85). Is basically a mix of the post-programming in programing phase #1 and the entire programming phase #2.
- **Check if application alive –** A test function which is used to check if there is a valid application after the performed test case by checking if the ECU sends periodic messages on the CAN-channel.

If one of the test cases fails during one of the sub-steps in the flow in Figure 4-11 the “Download with errors on all blocks except SBL” test case will perform a ECU Reset and check if the ECU is responsive, if the ECU is unresponsive then a rescue mission of the ECU will be performed by switching off the power to the ECU and turn it on again and send Diagnostic Session Control UDS-request in order to catch the ECU on boot. A normal programming will then be performed with no errors to reset the ECU to a functional state before starting the next test case. This sequence is shown in Figure 4-12.

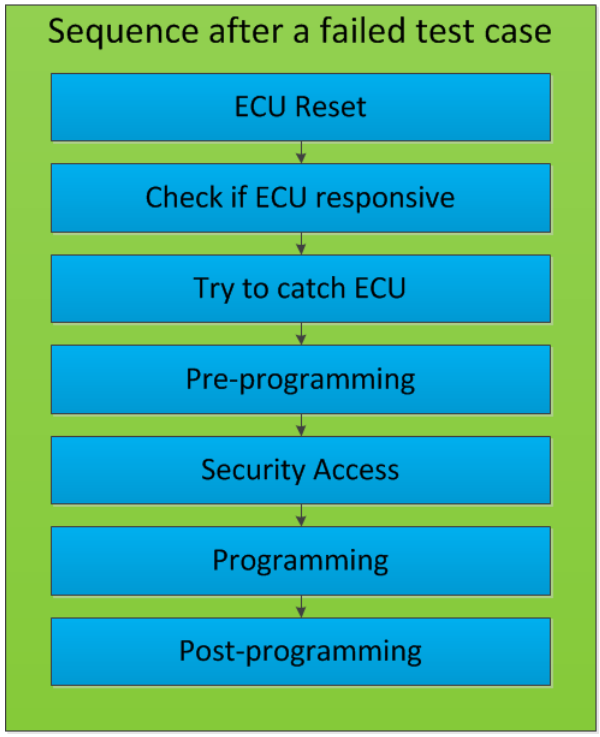

Figure 4-12: Sequence, which is run after a failed test case in the commonly used test case "Download with errors on all blocks except SBL".

A similar sequence to the one depicted in Figure 4-12 will also be performed if the check if the application is alive step fails in the sequence depicted in Figure 4-11 even if the test case was passed. This is done in order to ensure that the previous test case will not affect the following test case.

Other commonly used test cases

Another test, which can be commonly used, is the test case "Download only SBL and second block" which simply downloads one of the blocks other than the SBL and checks if the ECU will handle this. This test case is created by removing the test functions for the download of the third to the sixth block from the sequence in Figure 4-11 as can be seen in Figure 4-13. Another test case based on this method is the "Download SBL then other blocks in reverse order" to check if the ECU handles this. The user can also change which of the downloaded blocks should be injected with errors simply by modifying the parameter, which is sent in the sub-steps.

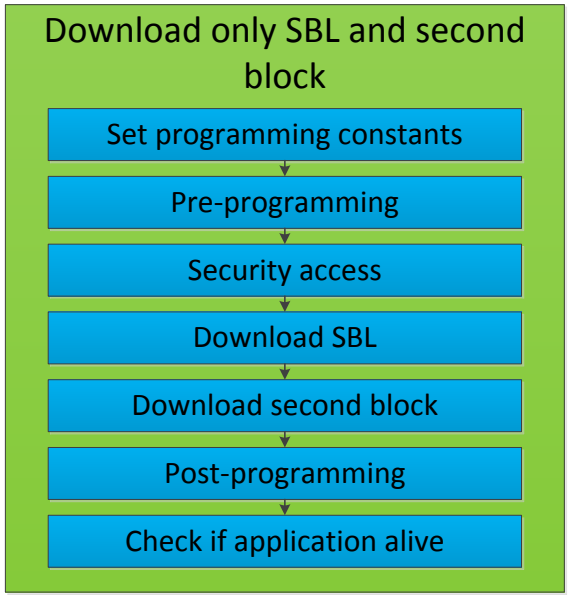

Figure 4-13: Flow chart for the test case of only loading SBL and second block for the test case "Download only SBL and second block".

5. Evaluation

5.1 Flash test cases

The flash test cases implemented in this project are focused on testing the generic requirements on a full software loading sequence specified in the UDS-standard. The tests implemented and tested in this project are explained in detail in the Appendix. These flash test cases cover requirements identified in the UDS-standard which should be accepted by the ECU, such as accepting duplicate Transfer Data Requests, but also covers areas where the ECU should be able to identify sequence errors and wrongly transmitted data. Some of the tests also cover recommendations in the UDS rather than strict requirements, such as the need to support different maximum lengths for Transfer Data requests.

In this project four ECU:s have been tested in all of the implemented test cases. A summary of the results from these tests is presented in the list below.

- Two ECU:s passed all the implemented test cases.
- One ECU failed on 6 of the 22 test cases or 27 % of the implemented test cases.
- One ECU failed on 7 out of the 22 test cases or 32 % of the implemented test cases.

A few notes to add to these results are that if tests that covers one area fails then it is likely that other tests will fail as well as they cover a related requirement. Also some of the tests included are recommendations in the UDS-standard rather than strict requirements.

5.2 Automatically generated test reports

In order to efficiently test and evaluate flash test cases on the software loading sequence there is a need for automatically generated test reports so that a test

engineer can analyze the downloading sequence once all tests have been run. Running all the test cases implemented in this project in sequence will take between 20 and 40 minutes depending on which ECU-project is being tested. If each of these tests had to be started and monitored manually by a test engineer then it would require a lot of time, whereas if when there is an automatic evaluation sequence performed on the software loading sequence then the most common errors can easily be complied into an automatically generated test report. An example of the overview given to the test engineer after all test have been performed on the current ECU is given in Figure 5-1. Example of an evaluation done in the test report is depicted in Figure 5-2.

Statistics

Overall number of test cases	22	
Executed test cases	22	100% of all test cases
Not executed test cases	0	0% of all test cases
Test cases passed	22	100% of executed test cases
Test cases failed	0	0% of executed test cases

Warnings occured during test execution.

Test Case Results

1	1	Subtract one byte from max block length	pass	
2	2	Set max block Length to 128 bytes	pass	
3	3	Send blockSequenceCounter twice on first block	pass	
4	4	Send blockSequenceCounter twice with error on the second block	pass	
5	5	Send blockSequenceCounter twice on all message	pass	
6	6	Send blocksequence fifty times on first request	pass	
7	7	Send wrong blockSeqeuenceCounter on first block	pass	
8	8	Send wrong blockSequenceCounter after rollover	pass	
9	9	Send wrong blockSequenceCounter after rollover and set block length to 128 bytes	pass	
10	10	Modify last byte in last transferdata request	pass	warning
11	11	Modify first byte in first transferdata request	pass	warning
12	12	Send empty block on first block	pass	warning
13	13	Send empty transferData request after last block	pass	warning
14	14	Send more data by adding dummy block	pass	warning
15	15	Send less data by removing one block	pass	warning
16	16	Send more data than specified	pass	warning
17	17	Send less data than specified	pass	warning
18	18	Send incorrect address and length identifier	pass	warning
19	19	Modify memoryAddress in request download	pass	warning
20	20	Send second file-block twice	pass	
21	21	Download only SBL and second file-block	pass	
22	22	Download file blocks in reverse order	pass	

Figure 5-1: Example of the overview the test engineer is given of the flash tests run on the software loading sequence by the automatically generated test reports in CANoe. A warning in the test report is usually sent when the ECU is unresponsive or there is

no valid application, but is an acceptable outcome in the current test case (see Figure 0-3 in Appendix).

5. DownloadSecondBlock - : Passed			
429.745581	iDCSendRequest	Sending 34 00 44 00 C1 00 00 00 03 DC 00	-
430.325487	iDCTestWaitForDiagResponse	Received 74 20 0F FF	-
430.325487	Info	Sending blockSequenceCounter = 2 on first block	-
430.325487	iDCSendRequest	Sending 36 02 with 4093 data bytes	-
430.665844	iDCTestWaitForDiagResponse	Received 7F 36 73	-
430.665844	Info	ECU noticed sequence error in blockSequenceCounter	-
430.665844	Diagnosis	Response received with NRC 0x73 (wrongBlockSequenceCounter)	pass
430.665844	iDCSendRequest	Sending 36 01 with 4093 data bytes	-
431.112653	iDCTestWaitForDiagResponse	Received 76 01	-
431.112653	iDCSendRequest	Sending 36 02 with 4093 data bytes	-
431.562605	iDCTestWaitForDiagResponse	Received 76 02	-

Figure 5-2: Example of the evaluation done in an automatically generated test report on the test case “Send wrong first blockSequenceCounter” (see Appendix)

6. Conclusions

This project has provided the software-testing department at BW-PDS the possibility of running test cases on a full UDS-based software loading sequence and thereby extending the test coverage to cover many of the requirements specified in ISO14229-1. This will enable the testing of both internally and externally developed ECU bootloader software. By being able to test and verify that the requirements specified in UDS are fulfilled, BW-PDS can deliver a final product that is more stable and follows the main international standard used for the software loading sequence in the road vehicles category.

In the final part of this project a large part of the time was spent on implementing a way to run automated test cases, which enables running test cases on the flashing sequence without the need for user intervention before all tests are completed saving a lot of unnecessary work. Then by analyzing the automatically generated test reports the user can make an informed judgment if the ECU follows the specification which the current test cases was designed to test.

During this project the advantages of having a customizable software loading sequence to test the both the requirements in the UDS-standard and emulate their customers flashing sequence have become apparent.

The flash test cases which, have been implemented and tested on both internally and externally developed ECU bootloader software, have shown that testing in this area is still to some degree limited and there are still issues present even in commercial products.

6.1 Future work

This project was never meant to be a final solution but instead provide a framework which could be expanded to fit the needs which arise during testing of the ECU bootloader software and thereby over time increase the test coverage as more and more experience is gained of the common issues. The focus in this

project has been to implement tests on generic requirements identified in the UDS-standard in the main software loading sequence, which had not been possible for the software testing department to test before this project as there was no way to perform a full software loading sequence in CANoe.

By focusing on the generic requirements the tests which were implemented could be run on several different ECU:s and thereby quickly increasing the test coverage across a wide range of projects. The future work following this project will contain implementing vendor-specific test cases, identifying more generic requirements and more test cases on the identified requirements. More tests are needed on both newly identified requirements as well as those currently identified, as there are several ways to test a single requirement that may present different results.

There are also other areas which could be tested, for example how external conditions would impact the software loading sequence by simulating the external conditions which can occur by utilizing the capabilities of CANoe. During this project the support for a few customer projects at BW-PDS have been omitted from the current solution as the most important projects were prioritized and focus was in some part to show the possibility of performing tests on a vendor-specific software loading sequence in a unified way. In future work more vendors could be supported within this framework relatively easy because of the standardization effort carried out by the International Organization for Standardization (ISO) to present the UDS-standard for road vehicles ISO14229-1.

7. References

[1] BorgWarner, “BorgWarner Official Presentation,” Landskrona, Sweden, 2014.

[2] M. Rings and P. Phillips, "Adding Unified Diagnostic Services over CAN to an HIL Test System" *SAE Technical Paper 2011-01-0454 ,* 2011, doi:10.4271/2011-01-0454.

[3] Vector Informatik GmBH, "Learning Module CAN" [Online]. Available: http://elearning.vector.com/index.php?wbt_ls_kapitel_id=1329975&root=378422&seite=vl_can_introduction_en. [Accessed 12 November 2015].

[4] National Instruments, "FlexRay Automotive Communication Bus Overview" 21 August 2009. [Online]. Available: http://www.ni.com/white-paper/3352/en/#toc1. [Accessed 12 November 2015].

[5] E. Mayer, "Serial Bus Systems in the Automobile - Part 2: Reliable data exchange in the automobile with CAN" Vector GmBh, December 2006. [Online]. Available: http://elearning.vector.com/portal/medien/cmc/press/PTR/SerialBusSystems_Part2_ElektronikAutomotive_200612_PressArticle_EN.pdf. [Accessed 20 November 2015].

[6] Vector Informatik GmBh, "Learning Module FlexRay" [Online]. Available: http://elearning.vector.com/index.php?wbt_ls_kapitel_id=1330154&root=378422&seite=vl_flexray_introduction_en. [Accessed 24 November 2015].

[7] Vector Informatik GmBh, "Learning module LIN" [Online]. Available: http://elearning.vector.com/index.php?wbt_ls_kapitel_id=1330149&root=378422&seite=vl_lin_introduction_en. [Accessed 8 January 2016].

[8] MOST Cooperation, “Motivation for MOST” [Online]. Available: http://www.mostcooperation.com/technology/introduction/. [Accessed 8

January 2016].

[9] Motorola Inc, Motorola M68000 Family Programmer's Reference Manual Motorola Inc, Schamburg, United States, 1992, pp. C-1 - C8.

[10] “Motorola's M6800 microcomputer system, which can operate from a single 5-volt supply, is moving out of the sampling stage and into full production” *Electronics,* pp. 114-115, 26 December, 1974 Vol.47 No. 26.

[11] V. Bordyk, "Analysis of software and hardware configuration" Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden, 2012.

[12] SB-Projects, “Intel HEX format” SB-Projects, [Online]. Available: http://www.sbprojects.com/knowledge/fileformats/intelhex.php. [Accessed 18 December 2015].

[13] Wikipedia, “EEPROM” 7 December 2015. [Online]. Available: https://en.wikipedia.org/wiki/EEPROM. [Accessed 19 February 2016].

[14] Wikipedia, “Flash memory” 6 February 2016. [Online]. Available: https://en.wikipedia.org/wiki/Flash_memory. [Accessed 19 February 2016].

[15] Vector Informatik GmbH, "ECU Development & Test with CANoe" 2015. [Online]. Available: http://vector.com/vi_canoe_en.html. [Accessed 11 November 2015].

[16] Vector CANtech, Inc., “Programming with CAPL” 14 December 2004. [Online]. Available: http://vector.com/portal/medien/vector_cantech/faq/ProgrammingWithCAPL.pdf. [Accessed 12 November 2015].

[17] Vector Informatik GmBh, "Include Files: Overview" in *CAPL Introduction*, Stuttgart, Germany, Vector Informatik GmBh, 2016.

[18] National Instruments, “ECU Designing and Testing using National Instruments Products,” 7 November 2009. [Online]. Available: http://www.ni.com/white-paper/3312/en/. [Accessed 19 November 2015].

[19] International Organization for Standardization, “Road vehicles - Unified diagnostic services (UDS) - Part 1: Specification and Requirements (ISO-14229-1:2013, IDT)” Swedish Standards Institute (SIS), Stockholm, Sweden, 2013.

[20] International Organization for Standardization, “Road vehicles - Diagnostic communication over Controller Area - Network (DoCAN) - Part 2: Transport protocol and network layer services (ISO 15765-2:2011, IDT)” Swedish Standards Institute, Stockholm, Sweden, 2011.

[21] J. G. D. Pehrsson, “Bootloader with reprogramming functionality for electronic control units in vehicles: Analysis, design and Implementation” Jönköping University College, School of Engineering, Jönköping, Sweden, 2012.

[22] AUTOSAR, “AUTOSAR - Enabling Innovation” AUTOSAR, 2014. [Online]. Available: http://www.autosar.org/. [Accessed 23 February 2016].

[23] A. Karlsson, “JTAG-Opimisation in CANoe” Department of Automatic Control, Lund, Sweden, 2014.

Appendix

Test design

General design

The test cases are divided into functionality groups, correlating to the requirements [19], as below:

- Software loading sequence – Services: 0x34, 0x36, 0x37
- P2 and P2 extended timings

All tests are implemented in CAPL and executed on a VT system using CANoe.

Any parameter, state or environment property not mentioned in a test case should be at its default setting.

Software loading sequence – Services: 0x34, 0x36, 0x37

The transfer of data in a flashing sequence is initiated by a Request Download request (0x34), followed by Transfer Data request (0x36) and terminated by a Request Transfer Exit request (0x37), how these requests are formulated are governed by the UDS-standard [19]. These test cases aim to test that the ECU will only accept correctly formulated requests and accept some variations that are permitted within in the UDS-standard [19].

Test case design

The test cases will be categorized into a few main categories: test cases on the Request Download request, test cases on the Transfer Data request, and general test cases on the software loading sequence.

Test cases on Request Download UDS-requests

These test cases modify the parameters in the 0x34 request: Sending less/more data than specified by modifying the memorySize [19, p.271] in a 0x34 request. Sending incorrect addressAndLengthIdentifier[19, p.271] in the 0x34 request. Sending the wrong memory address by modifying the memoryAddress [19, p.271] in the 0x34 request.

Send more data than requested memorySize

Request to download one (1) byte less in the memorySize parameter [19, p.271], than the total data block size. Should trigger NRC in RequestTransferExit [19, p.286]. (Extreme case: the last byte is sent in its own frame then NRC 24/71 could be trigged in TransferData [19, p.283]).

Send less data than requested memorySize

Request to download one (1) byte more in the memorySize parameter [19, p.271], than the total data block size. Should trigger NRC in RequestTransferExit [19, p.286].

Send incorrect address in addressAndLengthIdentifier

If the vehicle manufacturer uses 4 bytes (addressAndLengthIdentifier =0xX4) to represent the memoryAddress then request a representation using 1 bytes (addressAndLengthIdentifier =0xX1) but still sending 4 bytes of memoryAddress and vice versa. Should trigger NRC in RequestDownload [19, p.273].

Send modified memoryAddress

Add one (1) byte to start address or block number (block identifier) written to the memoryAddress of the block to be downloaded. Should trigger NRC in Request Download and/or Request Transfer Exit [19, p.273].

Test cases on Transfer Data UDS-requests

These test cases modify the parameters in the 0x36 request. Test cases on the blockSequenceCounter [19, p.282]. Test cases on the maxNumberOfBlockLength [19,p.277] received from the RequestDownload positive response. General test cases on modifying the binary data.

BlockSequenceCounter test cases

Test cases on the blockSequenceCounter [19, p.282].

Send blockSequenceCounter twice on first block

Send the same Transfer Data request with the same blockSequenceCounter on first Transfer Data Request. Should not generate any NRC according to [19, p.281].

Send blockSequenceCounter twice with error on block 2.

Send the same blockSequenceCounter twice with correct data in the first message and modified data in the next request. It is recommended that the ECU does not write the data if it has already received the current blockSequenceCounter. This will check if it does that or not [19, p.281]. Could generate incorrect checksum if the second message is written to EEPROM or flash memory.

Send blockSequenceCounter twice on all messages

Send the same Transfer Data request with the same blockSequenceCounter. Should not generate any NRC according to [19, p.281].

Send blockSequenceCounter fifty times on first on first block

Send the same Transfer Data request with the same blockSequenceCounter fifty times (arbitrary number). Should not generate any NRC according to [19, p.281].

Send wrong first blockSequenceCounter

Send blockSequenceCounter = 2 on the first Transfer Data-request instead of one (1) [19,p.280]. Should generate a NRC in Transfer Data [19, p.283].

Send wrong blockSequenceCounter after rollover

Send blockSequenceCounter = 0x01 after rollover from 0xFF instead of blockSequenceCounter = 0x00 [19, p. 280]. Should generate NRC in Transfer Data [19, p.283]. Note: requires a large data block / small maxNumberOfBlockLength to achieve a rollover from 0xFF.

Send wrong blockSequenceCounter after rollover maxBlockLength 128

Send blockSequenceCounter = 0x01 after rollover from 0xFF instead of blockSequenceCounter = 0x00 [19, p. 280]. Should generate NRC in Transfer Data [19, p.283]. If the ECU can handle a 128 byte Transfer Data request, then this will generate rollovers of blockSequenceCounters more frequently than previous test case.

MaxNumberOfBlockLength test cases

Test cases on the maxNumberOfBlockLength [19,p.277] received from the Request Download positive response. There is no actual requirement in ISO-14229-1 that Transfer Data-request, which is shorter than specified by the maxNumberOfBlockLength, should be accepted. However, if they are not accepted then ECU should send a NRC in Transfer Data [19, 283]. The last Transfer Data-request may however be shorter which should be accepted by the ECU according to the UDS-standard [19, p.277].

Subtract one byte from max block length

Send Transfer Data request, which is one (1)byte shorter than what is specified by the maxNumberOfBlockLength. Possible NRC in Transfer Data if not supported [19, p.283].

Set max block length to 128

Send Transfer Data request with a set length less than MaxNumberOfBlockLength. Possible NRC in Transfer Data if not supported [19, p.283].

General test cases on Transfer Data requests

Modify last byte of the binary data

Modifying the last data byte of the binary data should trigger a checksum error in the respective vehicle manufacturer flashing sequence.

Modify first byte of the binary data

Modifying the last data byte of the binary data should trigger a checksum error in the respective vehicle manufacturer flashing sequence.

Send empty block on first block

Send a Transfer Data Request with no data. Should trigger NRC in Transfer Data due to minimum length check [19, p.283]. This message should at least not be accepted.

Send empty block after last block

Send a Transfer Data Request with no data. Should trigger NRC in Transfer Data in minimum length check [19, p.283]. (Could however trigger NRC 24 [19, p.283] if sent as last Transfer Data UDS-request or NRC 31 if length not supported but NRC 13 should be triggered first according to evaluation sequence. This is however not followed by most manufacturers yet [19, s.284]).

Send more data by adding dummy block

Add one byte (0xFF) to the file-block, which is being downloaded. ECU should notice the error at least on Transfer Exit Request (0x37) if not earlier and send NRC.

Send less data by removing one byte

Removing last byte from the file-block, which is being downloaded. ECU should notice the error at least on Transfer Exit Request (0x37) if not earlier and send NRC.

7.1 P2 and P2 extended timings

7.1.1 Test cases on P2/P2 extended

Most timing failures regarding the P2/P2 extended [19, s.41] seem to occur during Routine Control service [19, p.260] based on observations.

Refer to ISO 14229-2 for further details on P2Server and P2*Server [19, p.41].

Test step warning if P2/P2 extended are exceeded by more than 10 %

Send a test step warning if the P2/P2 extended timings are exceeded by more than 10 % (arbitrary number).

Test Step Fail If P2/P2 extended are exceeded by more than 10 % and 100 ms extra wait time

Send a test step fail if the P2/P2 extended timings are exceeded by more than 10 % (arbitrary number) and an extra wait time of 100 ms (arbitrary number).

Test case	Accepted NRC 0x34	Non-acceptable NRC 0x34	Accepted NRC 0x36 (Bold = special eval)	Non-acceptable NRC 0x36
TransferData *- MaxNumberOfBlockLength Testcases*				
SUBTRACT_ONE_BYTE_FROM_MAX_BLOCK_LENGTH	-	-	0x24, 0x71, 0x72	-
SET_MAX_BLOCK_LENGTH_TO_128	-	-	0x24, 0x71, 0x72	-
TransferData *- blockSequenceCounter Testcases*				
SEND_BLOCK_SEQUENCE_COUNTER_TWICE_ON_FIRST_BLOCK	-	-	No NRC	0x24, 0x73, 0xXX
SEND_BLOCK_SEQUENCE_COUNTER_TWICE_ERROR_ON_BLOCK_2	-	-	No NRC	0x24, 0x73, 0xXX
SEND_BLOCK_SEQUENCE_COUNTER_TWICE_ON_ALL_MESSAGES	-	-	No NRC	0x24, 0x73, 0xXX
SEND_BLOCK_SEQUENCE_COUNTER_FIFTY_TIMES_ON_FIRST_BLOCK	-	-	No NRC	0x24, 0x73, 0xXX
SEND_WRONG_FIRST_BLOCK_SEQUENCE_COUNTER	-	-	**0x24, 0x73**	-
SEND_WRONG_BLOCK_SEQUENCE_COUNTER_AFTER_ROLLOVER	-	-	**0x24, 0x73**	-
SEND_WRONG_BLOCK_SEQUENCE_COUNTER_AFTER_ROLLOVER_MAX_BLOCK_LENGTH_128	-	-	0x24, **0x24**, 0x71, 0x72, **0x73**	-
TransferData *- General Testcases*				
MODIFY_LAST_BYTE_IN_LAST_TRANSFER_DATA_REQUEST	-	-	0x24, 0x72	-
MODIFY_FIRST_BYTE_IN_FIRST_TRANSFER_DATA_REQUEST	-	-	0x24, 0x72	-
SEND_EMPTY_BLOCK_ON_FIRST_BLOCK	-	-	**0x13**, **0x24**, 0x24, **0x31, 0x71**	-
SEND_EMPTY_BLOCK_AFTER_LAST_BLOCK	-	-	**0x13, 0x24 ,** 0x24, 0x31, **0x31**	-
SEND_MORE_DATA_BY_ADDING_DUMMY_BYTE	-	-	0x13, 0x24, 0x71, 0x72, 0x73	-
SEND_LESS_DATA_BY_REMOVING_ONE_BYTE	-	-	0x13, 0x24, 0x71, 0x72, 0x73	-
RequestDownload *- General testcases*				
SEND_MORE_DATA_THAN_REQUESTED_MEMORY_SIZE	0x24	-	0x13, 0x24, 0x71, 0x72	-
SEND_LESS_DATA_THAN_REQUESTED_MEMORY_SIZE	0x24	-	0x13, 0x24, 0x71, 0x72	-
SEND_INCORRECT_ADDRESS_IN_ADDRESS_AND_LENGTH_ID	0x13	-	0x24, 0x71	-
SEND_MODIFED_MEMORY_ADDRESS	0x31	-	0x24	-
Software loading sequence *- General Testcases*				
SEND_SECOND_FILE_BLOCK_TWICE	No NRC	-	No NRC	-
DOWNLOAD_ONLY_SBL_AND_SECOND_BLOCK	No NRC	-	No NRC	-
DOWNLOAD_FILE_BLOCKS_IN_REVERSE_ORDER	No NRC	-	No NRC	-

Figure 0-1: List over accepted NRC:s in evaluation of services Request Download (0x34) and Transfer Data (0x36). There is an additional evaluation on service 0x36 which evaluates "special" events, which generally only occur a limited amount of times, the NRC:s corresponding to the special evaluation are marked with bold text. Note that there may be more NRC:s which are acceptable, this list only contains those which have occurred during testing. Therefore consider adding acceptable NRC:s to evaluation as they occur.

Test case	Accepted NRC 0x37	Non-acceptable NRC 0x37	Positive response on 0x34, 0x36, 0x37	Comment
TransferData *- MaxNumberOfBlockLength Testcases*				
SUBTRACT_ONE_BYTE_FROM_MAX_BLOCK_LENGTH	-	-	Accepted	BlockLength might not supported by manufacturer
SET_MAX_BLOCK_LENGTH_TO_128	-	-	Accepted	BlockLength might not supported by manufacturer
TransferData *- blockSequenceCounter Testcases*				
SEND_BLOCK_SEQUENCE_COUNTER_TWICE_ON_FIRST_BLOCK	No NRC	No NRC	Should be accepted	
SEND_BLOCK_SEQUENCE_COUNTER_TWICE_ERROR_ON_BLOCK_2	No NRC	No NRC	Should be accepted	
SEND_BLOCK_SEQUENCE_COUNTER_TWICE_ON_ALL_MESSAGES	No NRC	No NRC	Should be accepted	
SEND_BLOCK_SEQUENCE_COUNTER_FIFTY_TIMES_ON_FIRST_BLOCK	No NRC	No NRC	Should be accepted	
SEND_WRONG_FIRST_BLOCK_SEQUENCE_COUNTER	No NRC	No NRC	Not-accepted on 0x36	
SEND_WRONG_BLOCK_SEQUENCE_COUNTER_AFTER_ROLLOVER	No NRC	No NRC	Not-accepted on 0x36	
SEND_WRONG_BLOCK_SEQUENCE_COUNTER_AFTER_ROLLOVER_MAX_BLOCK_LENGTH_128	No NRC	0x24	Not-accepted on 0x36	BlockLength might not be supported by manufacturer
TransferData *- General Testcases*				
MODIFY_LAST_BYTE_IN_LAST_TRANSFER_DATA_REQUEST	0x24	-	Accepted	
MODIFY_FIRST_BYTE_IN_FIRST_TRANSFER_DATA_REQUEST	0x24	-	Accepted	
SEND_EMPTY_BLOCK_ON_FIRST_BLOCK	0x24	-	Not-accepted on 0x36	
SEND_EMPTY_BLOCK_AFTER_LAST_BLOCK	0x24	-	Not-accepted on 0x36	
SEND_MORE_DATA_BY_ADDING_DUMMY_BYTE	0x22, 0x24	-	Not-accepted on 0x37	
SEND_LESS_DATA_BY_REMOVING_ONE_BYTE	0x22, 0x24	-	Not-accepted on 0x37	
RequestDownload *- General testcases*				
SEND_MORE_DATA_THAN_REQUESTED_MEMORY_SIZE	0x22, 0x24	-	Not-accepted on 0x37	
SEND_LESS_DATA_THAN_REQUESTED_MEMORY_SIZE	0x22, 0x24	-	Not-accepted on 0x37	
SEND_INCORRECT_ADDRESS_IN_ADDRESS_AND_LENGTH_ID	0x24	-	Not-accepted on 0x34	
SEND_MODIFED_MEMORY_ADDRESS	0x24	-	Not-accepted on 0x37/34?	
Software loading sequence *- General Testcases*				
SEND_SECOND_FILE_BLOCK_TWICE	No NRC	-	All Should be accepted	
DOWNLOAD_ONLY_SBL_AND_SECOND_BLOCK	No NRC	-	All Should be accepted	Requires that other blocks are valid before
DOWNLOAD_FILE_BLOCKS_IN_REVERSE_ORDER	No NRC	-	All Should be accepted	

Figure 0-2: List over accepted NRC:s in evaluation of service Request Transfer Exit (0x37). Note that there may be more NRC:s which are acceptable, this list only contains those which have occurred during testing. Therefore consider adding acceptable NRC:s to evaluation as they occur.

Test case	CheckMemory/ProgrammingDependencies/ValidApp	EvaluateChecksumFromRequestTransferExit	EvaluateCheckIfAppAlive	EvaluateECU_NonResponsive
TransferData - MaxNumberOfBlockLength Testcases				
SUBTRACT_ONE_BYTE_FROM_MAX_BLOCK_LENGTH	Should be correct	Should be correct	Should be alive	Should be responsive
SET_MAX_BLOCK_LENGTH_TO_128	Should be correct	Should be correct	Should be alive	Should be responsive
TransferData - blockSequenceCounter Testcases				
SEND_BLOCK_SEQUENCE_COUNTER_TWICE_ON_FIRST_BLOCK	Should be correct	Should be correct	Should be alive	Should be responsive
SEND_BLOCK_SEQUENCE_COUNTER_TWICE_ERROR_ON_BLOCK_2	Should be correct	Should be correct	Should be alive	Should be responsive
SEND_BLOCK_SEQUENCE_COUNTER_TWICE_ON_ALL_MESSAGES	Should be correct	Should be correct	Should be alive	Should be responsive
SEND_BLOCK_SEQUENCE_COUNTER_FIFTY_TIMES_ON_FIRST_BLOCK	Should be correct	Should be correct	Should be alive	Should be responsive
SEND_WRONG_FIRST_BLOCK_SEQUENCE_COUNTER	Should be correct	Should be correct	Should be alive	Should be responsive
SEND_WRONG_BLOCK_SEQUENCE_COUNTER_AFTER_ROLLOVER	Should be correct	Should be correct	Should be alive	Should be responsive
SEND_WRONG_BLOCK_SEQUENCE_COUNTER_AFTER_ROLLOVER_MAX_BLOCK_LENGTH_128	Should be correct	Should be correct	Should be alive	Should be responsive
TransferData - General Testcases				
MODIFY_LAST_BYTE_IN_LAST_TRANSFER_DATA_REQUEST	Should be incorrect	Should be incorrect	Acceptable if not alive	Acceptable if not responsive
MODIFY_FIRST_BYTE_IN_FIRST_TRANSFER_DATA_REQUEST	Should be incorrect	Should be incorrect	Acceptable if not alive	Acceptable if not responsive
SEND_EMPTY_BLOCK_ON_FIRST_BLOCK	Should be incorrect	Could be incorrect	Acceptable if not alive	Should be responsive
SEND_EMPTY_BLOCK_AFTER_LAST_BLOCK	Should be incorrect	Could be incorrect	Acceptable if not alive	Should be responsive
SEND_MORE_DATA_BY_ADDING_DUMMY_BYTE	Should be incorrect	Could be incorrect	Acceptable if not alive	Acceptable if not responsive
SEND_LESS_DATA_BY_REMOVING_ONE_BYTE	Should be incorrect	Could be incorrect	Acceptable if not alive	Acceptable if not responsive
RequestDownload - General testcases				
SEND_MORE_DATA_THAN_REQUESTED_MEMORY_SIZE	Should be incorrect	Could be incorrect	Acceptable if not alive	Acceptable if not responsive
SEND_LESS_DATA_THAN_REQUESTED_MEMORY_SIZE	Should be incorrect	Could be incorrect	Acceptable if not alive	Acceptable if not responsive
SEND_INCORRECT_ADDRESS_IN_ADDRESS_AND_LENGTH_ID	Should be incorrect	Could be incorrect	Acceptable if not alive	Acceptable if not responsive
SEND_MODIFED_MEMORY_ADDRESS	Should be incorrect	Could be incorrect	Acceptable if not alive	Acceptable if not responsive
Software loading sequence - General Testcases				
SEND_SECOND_FILE_BLOCK_TWICE	Should be correct	Should be correct	Should be alive	Should be responsive
DOWNLOAD_ONLY_SBL_AND_SECOND_BLOCK	Should be correct	Should be correct	Should be alive	Should be responsive
DOWNLOAD_FILE_BLOCKS_IN_REVERSE_ORDER	Should be correct	Should be correct	Should be alive	Should be responsive

Figure 0-3: List over evaluation of various checks, which are performed during the flashing sequence. The Routine Controls CheckMemory, Programming dependencies and CheckValidApplication will return a correct or incorrect result. This result is evaluated based on the test case, which has been run. Evaluation of checksum from Transfer Exit (0x37) will compare the checksums and evaluate base of the test case if it is correct. EvaluateCheckAppAlive will check if it is okay that the application is not running in the corresponding test case. EvaluateECU_NonResponsive will check if it is okay that the ECU is non-responsive in the corresponding test case.