|  |  |
| --- | --- |
| 199.jpg | HANTUNE |
|  | BitsChipsRC30Target_HANtune.jpg |
| Version 1.0  11/12/2012 | XCP master implementation |
|  | This document explains how the XCP protocol works and is implemented in HANtune.  By Aart-Jan van der Hoeven |

Copyright (c) 2020 [HAN University of Applied Sciences]

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

HANTUNE

XCP master implementation

Contents

[Summary 3](#_Toc341091056)

[1 Introduction 6](#_Toc341091057)

[2 Background 8](#_Toc341091058)

[3 XCP 1.0 protocol 10](#_Toc341091059)

[3.1 XCP features 10](#_Toc341091060)

[3.2 Communication modes 12](#_Toc341091061)

[3.3 XCP messages and communication flow 13](#_Toc341091062)

[4 XCP master tasks 15](#_Toc341091063)

[5 Driver interface 16](#_Toc341091064)

[6 Functional overview 18](#_Toc341091065)

[7 Commands 20](#_Toc341091066)

[7.1 STD - Standard commands 20](#_Toc341091067)

[7.2 CAL - Calibration commands 21](#_Toc341091068)

[7.3 PAG - Paging commands 21](#_Toc341091069)

[7.4 DAQ - Data acquisition and Stimulation commands 22](#_Toc341091070)

[7.5 PGM - Nonvolatile memory programming commands 23](#_Toc341091071)

[8 XCP information and settings 24](#_Toc341091072)

[8.1 General XCP settings 24](#_Toc341091073)

[8.2 Data acquisition settings 24](#_Toc341091074)

[9 Event channels 25](#_Toc341091075)

[10 DAQ lists 26](#_Toc341091076)

[11 Command List 28](#_Toc341091077)

[12 Error handling 30](#_Toc341091078)

[12.1 Table of error codes 30](#_Toc341091079)

[12.2 Error handling matrix 31](#_Toc341091080)

[13 Seed and Key mechanism 33](#_Toc341091081)

[13.1 ASAM Seed and Key mechanism 33](#_Toc341091082)

[13.2 Java Native Interface 34](#_Toc341091083)

[13.3 Implementation 35](#_Toc341091084)

[14 Limitations 36](#_Toc341091085)

[15 Functional tests 37](#_Toc341091086)

[16 Conclusion and recommendations 39](#_Toc341091087)

[References 40](#_Toc341091088)

[Appendix A - Settings in XCPinfo 41](#_Toc341091089)

[Appendix B - Settings in DAQinfo 44](#_Toc341091090)

[Appendix C - Settings for events 46](#_Toc341091091)

# Summary

What is the XCP protocol standard and how can it be implemented? A description is given to provide future developers of the XCP master device driver, HANtune and other related software a basic knowledge of the XCP standard and how to modify and interact with the developed XCP master device driver.

During development of an ECU application, the engineer may want to calibrate and measure several ECU parameters and may want to flash new software into the ECU. These functions are described by the XCP (Universal Calibration Protocol) protocol standard, written by ASAM. This standard defines the communication between master (tool) and slave (ECU) device, but also how these devices should take action in response to this communication.

Typical XCP features are:

* **Data acquisition**; The content of the slave device memory can be requested manually or by configuring a data acquisition list at the slave device, that is triggered by an (cyclic) event.
* **Data stimulation**; The control software at the slave device can be bypassed to perform calculations at an external computer. The slave device is only used for input and output.
* **Online memory calibration**; New data can be written to the slave device memory at runtime.
* **Data paging**; Multiple pages can be used to access the same data at the slave device. Different ECU and XCP pages are used for instance when transferring large amounts of calibration data.
* **Flash programming**; New software can be flashed into non-volatile memory at the slave device.

In this assignment, the XCP master was implemented in Java. Main aspects of this implementation are:

* The XCP protocol is split into a transport and protocol layer. These layers can be implemented as separate Java objects. This way the protocol layer is generic and can be used with different transport layers.
* Communication can be controlled by putting the XCP commands into a sequence (the command list). This sequence can be manipulated when an error is detected or when command do have to be repeated. All information like position, error code and number of repeats can be stored into this sequence.
* The command list is handled by a cyclic background process, which also performs the calls for sending a new message. This handling includes a nested decision matrix for manipulating the command list or exiting the sequence and returning the status. This decision matrix also includes the error handling matrix as defined by ASAM in the protocol standard.
* XCP information can be stored into Java data objects. This might be information obtained from the slave device, but also configuration information that has to be send to the slave device can be organized in special data objects. These data objects can be used for data exchange between the various layers.
* The Seed and Key mechanism for unlocking a specific resource can be implemented using a Java Native Interface between the Java program and the external provided DLL file.

DEFINITIONS AND ABBREVIATIONS

The following table gives an overview about the most commonly used definitions and abbreviations in this document.

|  |  |
| --- | --- |
| **Abbreviation** | **Description** |
| ASAM | Association for Standardization of Automation and Measuring Systems |
| ASAM-MCD1 | XCP standard by ASAM |
| ASAM-MCD2 | ASAP2 standard by ASAM |
| ASAP2 | ECU description database format defined by ASAP (Arbeitskreis zur Standardisierung von Applikationssystemen), the former name of ASAM |
| CAL | Calibration |
| calibration | modification of ECU parameters |
| CAN | Controller Area Network |
| CCP | CAN Calibration Protocol |
| CMD | Command message (CTO packet) |
| CTO | Command Transfer Object |
| cyclic | periodic, fixed rate |
| DAQ | Data acquisition (DTO packet) |
| DTO | Data Transfer Object |
| ECU | Electronic Control Unit |
| ERR | Error message (CTO packet) |
| EV | Event message (CTO packet) |
| HANtune | ECU Measurement and Calibration software tool |
| Javadoc | HTML (webpage) documentation generated out of the Java code comments |
| JNI | Java Native Interface |
| master | tool device |
| ODT | Object Description Table |
| PAG | XCP Paging |
| PGM | Flash programming |
| prescaler | value to divide a frequency to another frequency |
| RES | Response message (CTO packet) |
| RESUME mode | Automatically XCP (re)configuration after slave device start-up |
| SERV | Service message (CTO packet) |
| slave | ECU device |
| STIM | Data stimulation (DTO packet) |
| XCP | Universal Calibration Protocol |

# Introduction

During development of an ECU application, the engineer may want to calibrate and measure several ECU parameters and may want to flash new software into the ECU. These functions are described by the XCP (Universal Calibration Protocol) protocol standard, written by ASAM. This standard defines the communication between master (tool) and slave (ECU) device, but also how these devices should take action in response to this communication.

At HAN Automotive, multiple MATLAB Simulink blocksets are developed to generate target specific software executables out of a simulation model. Parallel to these activities a measurement and calibration tool (HANtune) is being developed to provide XCP communication with the ECU, based on an ECU description file (ASAP2). A XCP master device driver is written to provide XCP functionality as a separate layer to the HANtune software.

The purpose of this document is to give an answer to the following questions:

* What is the XCP protocol standard?
* How can the XCP protocol be implemented using Java?

This document gives future developers of the XCP master device driver, HANtune and other related software a basic knowledge of the XCP standard and how to modify and interact with the developed XCP master device driver. It will also inform people related to this project about the research and work that has been done on this subject. More detailed information about the XCP standard can be found in the “MCD1 – XCP 1.0” documentation provided by ASAM [3]. More detailed information about the internals of the XCP master device driver can be found in the code comments and generated Javadoc documentation. This document is mainly a result of summarizing the work being done, in this project, related to XCP, based on the documentation provided by ASAM. References are made when using other sources.

First, some background information is given in chapter 2. An overview of the XCP protocol is given in chapter 3. In chapter 4 the XCP related tasks of the master device are defined. Chapter 5 shows the interface that is defined between the available layers. Chapter 6 gives a functional overview of the internals of the protocol layer. An overview of the XCP commands is given in chapter 7. The settings that can be requested from the slave device are described in chapter 8. Chapter 9 gives a description of event channels. DAQ lists are explained in chapter 10. The command list mechanism is described in chapter 11. An overview of the error handling functionality is given in chapter 12. The Seed and Key mechanism is described in chapter 13. The limitations of the current implementation is given in chapter 14. The work that has been done one functional testing is described in chapter 14. Finally a conclusion is given in chapter 15.

# Background

Until 1992 different solutions have been used for developing, calibration, production and service of ECU hardware and software. In 1992 the CAN Calibration Protocol (CCP) was defined by ASAM to create a common tool for all stages of ECU development. This tool made the ECU compatible with different types of hardware and software. The ASAM group was founded by Audi, BMW, Mercedes-Benz, Porsche and Volkswagen [1; 2]. Later, several other manufacturers of hardware and software have joined the ASAM group to eliminate all the proprietary systems and interfaces that were developed by each company. Some of these manufacturers are for instance: DaimlerChrysler, dSPACE, ETAS, Bosch, Siemens and Vector [3].

Based on the experiences with CCP an improved and generalized version was developed in 2003 as the XCP protocol family. The “X” stands for the various transport layers that can be used for XCP communication. The XCP protocol family includes a generic protocol layer specification and several transport layer specifications. Some of these transport layers are for instance CAN, TCP/IP, USB and Flexray. This way it has become transport layer independent [5]. A visual representation of the protocol family can be seen in Figure 1.



Figure Protocol Layer with several Transport Layers

Common areas of use are [3; 4]:

* development of electronic control units (ECU)
* systems for functional and environmental tests of an ECU
* test systems and test stands for the controlled devices (combustion engines, gearboxes, suspension systems, climate control systems, body systems, anti-locking systems)
* on-board test and measurement systems of pre-series vehicles
* any non-automotive application of distributed electronic control systems

At HAN Automotive the use of XCP will mainly be focused at rapid control prototyping. As already mentioned, several target hardware specific blocksets for MATLAB Simulink are developed, to generate ready-to-use executables out of a simulation model. This executable also contains XCP slave support. Parallel to the generation of this executable, an ASAP2 database which describes the ECU will also be generated. This ASAP2 file contains for instance information about supported communication and parameter and signal properties. A measurement and calibration tool, like HANtune, can use this database file as an input. This input is used to set-up communication between master and slave device. An overview of this situation has been put into a scheme for better understanding, as can be seen in Figure 2. Basically , this is the same method that is used and described by Vector [6].

HANtune, the measurement and calibration tool, is built from scratch. An implementation of the XCP protocol family has to be written to allow XCP communication, because no ready-to-use implementations are available at the time of writing. The XCP implementation will be written in the Java programming language, as that is also the language that HANtune is written in. At first it will only support a basic XCP implementation, but will use the same separation between protocol and transport layer as given by ASAM. By using a modular design it will be easy to maintain and extend by future developers, but is also easy to use in software other than HANtune.

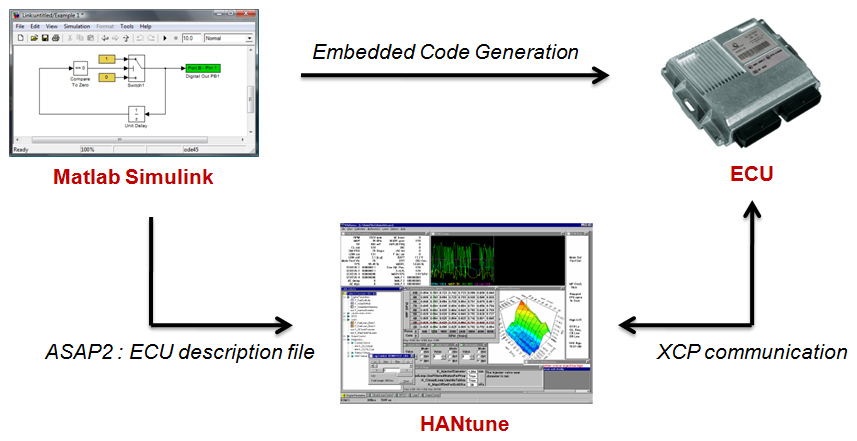


Figure Rapid control prototyping situation

Some of the first projects the software might be used are:

* Silent Motor Company – Hydrogen truck
* Burton – Electrification
* Combustion Analyzer

# XCP 1.0 protocol

This chapter will give a short summary of the XCP 1.0 protocol. However, for a more complete understanding of the standard it might be necessary to read the more detailed information in the XCP protocol documentation [3]. References to this information are added when necessary.

XCP is an universal standardized protocol for communication between a master (tool) and slave (ECU) device, but also describes how these devices should take action in response to this communication. These actions include functionality for calibration and measurements, but also for flash programming.

The XCP protocol basically is a single-master/single-slave type of communication. Any communication is always initiated by the master. The slave has to respond upon requests from the master with an appropriate response. It uses a ‘soft’ principal. The slave is not expected to send a direct response to the master.

## XCP features

XCP provides features that can be grouped into five feature groups. For each group a description is given and is told which part of the feature is implemented yet in the first XCP master driver version.

* **Data acquisition**  
  Data elements available at the slave device can be send to the master device. This is used to request current parameter values and to log and visualise signals. Data elements can be requested ‘manually’ by sending a direct request for the element to the slave device, but also by configuring a data acquisition list (DAQ list) at the slave device during session start-up. Such a DAQ list configuration might also be used in RESUME mode to automatically start data acquisition after start-up of the slave device. This might be handy when for instance doing cold-start measurements.  
    
  First implementation:
  + Perform ‘manual’ data acquisition
  + Perform DAQ list configurations
  + No RESUME mode support
* **Data stimulation**The control software which runs at the slave device might be bypassed using XCP. In that case, the master will read data using DAQ and uses these values for calculations in its own model. Calculated values out of that model are then returned back to the ECU by using STIM messages. This mechanism is used when tests and optimizations have to be made independent of the hardware and to perform calculations with less effort [4]. A schematic overview of bypassing is given in Figure 3.

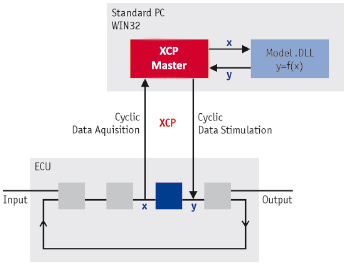


Figure Bypassing

First implementation:

* + No (explicit) data stimulation support
* **Online memory calibration**New data can be written to the memory of the ECU during runtime. The master has to send the memory address, data length and new data bytes when performing a calibration. The memory address and data length are obtained using the ASAP2 database, the new data bytes are user input or come from a calibration file.  
    
  First implementation:
  + Online memory calibration, but only in standard communication mode
* **Data paging**When performing online memory calibrations, it may happen that large amounts of data do have to be transferred to the slave device, for instance characteristic maps. While executing the calibration, the slave is also using the current available values for runtime calculations. Errors and unwanted situations might happen when calculations are made using old and new values. This is the reason for the data paging mechanism described by ASAM. For one part of data, a segment, multiple pages might by initialized containing the same data. The master is responsible for controlling the access of these pages. The master is able to switch the page the ECU is currently using for a specific segment, the same for XCP access. It is now possible to assign the ECU to one page and write new values to another page. When all calibrations are done, the active page for the ECU is switched to the modified page. This way data will always be coherent. A schematic overview is given in Figure 4.  
    
  First implementation:
  + No data paging support

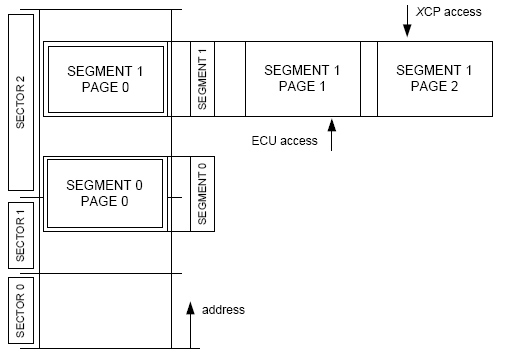
****

Figure Calibration Data Organisation

* **Flash programming**New software can be flashed into the slave device using XCP. Functions are defined for administration before flashing, performing the actual flashing and administration after flashing.  
    
  First implementation:
  + No flash programming support

## Communication modes

Three types of communication modes are defined:

* **Standard communication mode**Each request packet will be responded by a corresponding response or error packet. The master device may not send a new request until the response to the previous request has been received.
* **Block Transfer communication mode**To speed up long sequences, some commands may support block transfer mode. In this mode, the master and/or slave is allowed to send multiple subsequent messages. The block size and separation time may be limited though.
* **Interleaved communication mode**To speed up data transfer, in interleaved mode the master may already send the next request before having received the response on the previous request.

## XCP messages and communication flow

A XCP message consists of a XCP header, XCP packet and XCP tail. The content of the XCP header and tail are transport layer specific. The XCP packet is generic and has two possible types: Command Transfer Object (CTO) packet and Data Transfer Object (DTO) packet. See Figure 5.

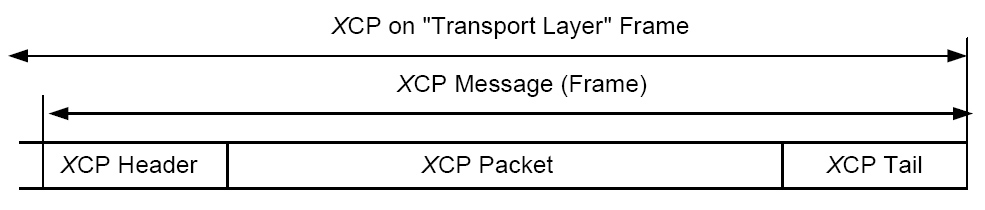


Figure XCP message format

CTO packets are used for transferring generic control commands. They are used for carrying out protocol commands (CMD), transferring command responses (RES), error (ERR) packets, event (EV) packets and for service request packets (SERV).

DTO packets are used for transmitting synchronous data acquisition data (DAQ) and for transmitting synchronous data stimulation data (STIM).

A command packet must always be answered by command response packet or an error packet. Event, service request and data acquisition packets are send asynchronously. An overview of the communication flow between master and slave device is given in Figure 6.

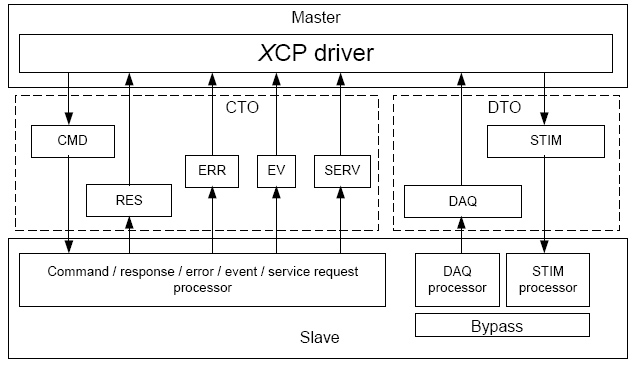


Figure XCP communication flow between master and slave device

The commands are grouped in standard, calibration, paging, data acquisition and simulation and programming command resources. These commands are explained in chapter 7. Resources can be protected by the slave device. The master first has to unlock this resource using a seed and key mechanism, before it can use the commands of this resource.

DAQ messages are generated using a DAQ list configuration. A DAQ list is a list of items referring to a memory address, that is send when triggered by an event channel. These DAQ lists are configured by the XCP master. The event channels are provided by the XCP slave and cannot be changed by the XCP master. More information about XCP event channels and DAQ lists can be found in chapter 10 and chapter 11.

# XCP master tasks

After the connection is made the master controls all communication between master and slave. In standard communication mode, every message is followed by a reply message from the slave containing data or error codes.

A general overview of the XCP master tasks:

* **Slave identification**Before starting any calibration, measurement or programming sequence, the master has to identify the slave device. This is done by requesting the slave identifier, but also by requesting protocol and transport layer versions and provided XCP support.
* **Session handling**The master is responsible for starting, stopping and resetting XCP sessions with the slave device.
* **Error handling**  
  Because the master is responsible for the communication, it is also responsible for proper error handling when the slave returns an error or when a connection timeout occurs. The master has to bring the slave in a well-defined state, before continuing the original command sequence.
* **Unlocking resources**Protected command resources have to be unlocked before the master is able to use them. The master has to use a seed and key mechanism to unlock a specific resource.
* **DAQ list configuration**The generation of DAQ messages depends on the DAQ list configuration that is active at the slave device. This configuration is set by the master device.
* **Data acquisition**The master has to convert the received DAQ messages to a format that is easy to use.
* **XCP paging**It might be necessary to use separate pages for XCP and ECU access. In this case, the master has to control the paging activities.
* **Calibrations**The master is responsible for sending the correct data for calibrating memory areas at the slave device.
* **Communication logging (optional)**All incoming and outgoing XCP data is logged into a logfile by the master device. This logfile is especially useful when debugging.

# Driver interface

The XCP master device driver consists of two separated parts. The XCP Protocol Layer in which all generic protocol handling takes place and the XCP Transport Layer in which all transport layer (CAN, Flexray, serial and more) specific handling takes place. These parts interact with each other and the (HANtune) Application Layer by various functions. An overview of this interface is given in Figure 7.

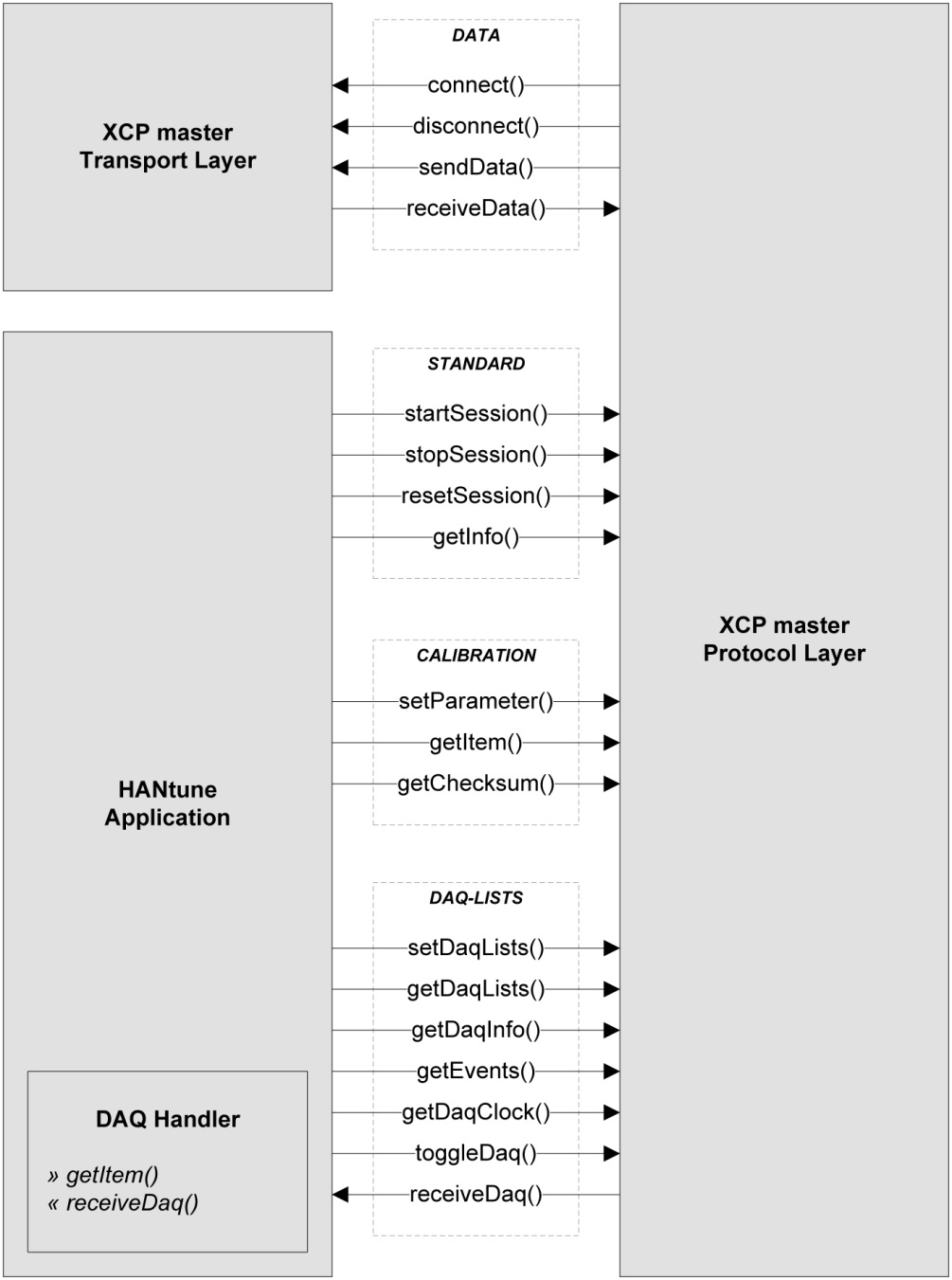


Figure XCP master device driver interface

The interface functions are grouped based on their related task. All Transport Layer interface functions are placed in the DATA group. These functions make it possible to connect and disconnect the slave device (network) and send and receive data over this connection. For now, only a CAN version of the XCP Transport Layer is written for the PEAK CAN-USB converter. However, other hardware versions are easy to implement, using the same interface functions and replacing the current XCP Transport Layer.

All common functions for starting, stopping and resetting a XCP session are gathered in the STANDARD group. The function for getting general XCP information and settings is also included in the STANDARD group.

All functions related to calibration activities are gathered in the CALIBRATION group. This group contains functions for setting a parameter, reading an item and requesting a checksum for a specified range.

All functions related to DAQ list activities are gathered in the DAQ-LIST group. This group contains functions for setting and getting a DAQ list configuration, requesting the available events, requesting DAQ list related information and settings, requesting the slaves clock value, start/stop/select a DAQ list and receiving of DAQ list messages from the slave device.

The DAQ handler, a subclass in the Application Layer, is responsible for handling the incoming DAQ list messages and manual requesting of items.

More information about internal set-up and data exchange between the specified layers is given in next chapters. For more detailed information about the mentioned functions, see the Javadoc documentation and Java code comments.

# Functional overview

A schematic overview of the internal set-up of the XCP master device driver is given in Figure 8.

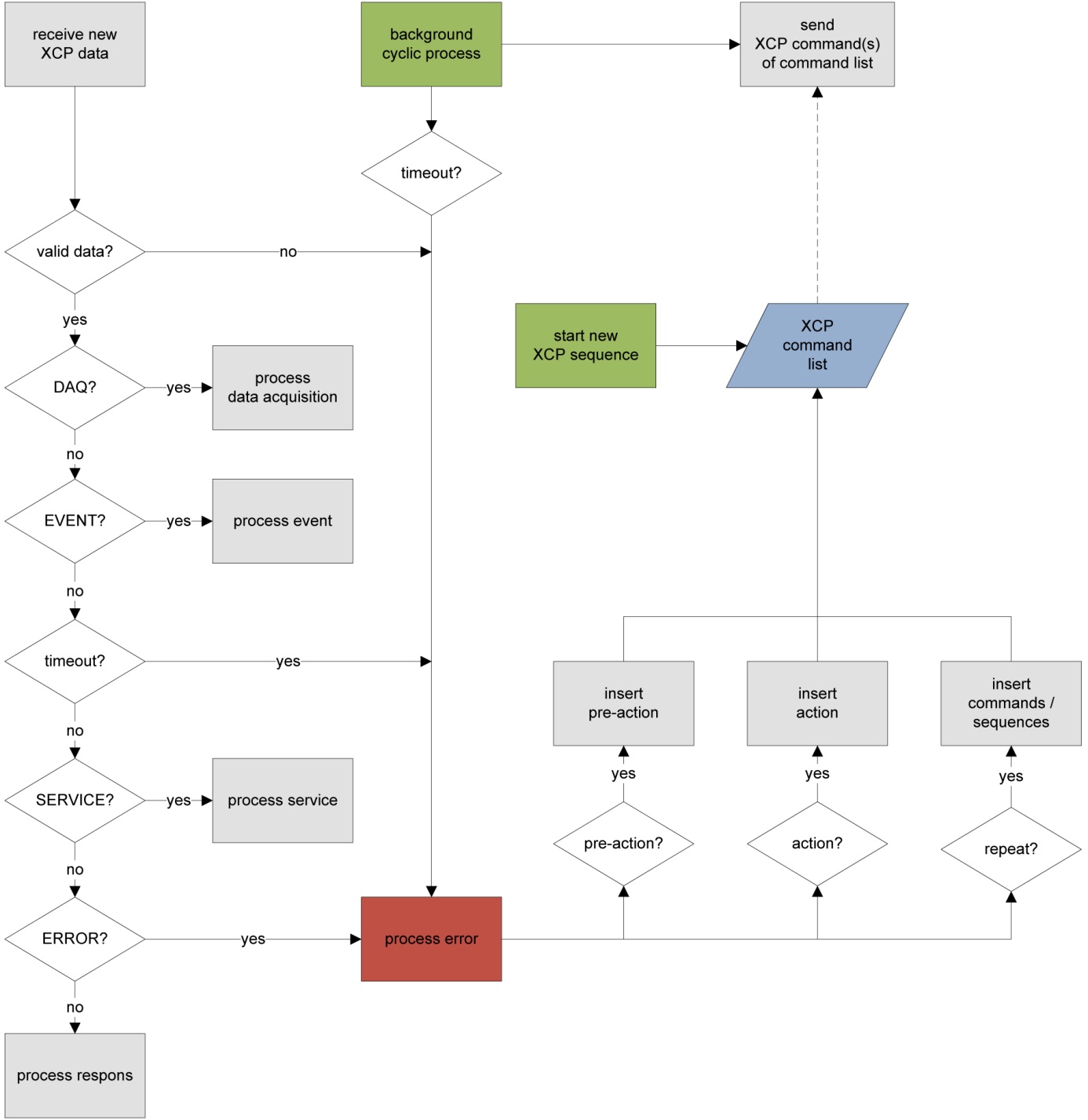


Figure XCP master driver functional overview

Central in the set-up is the XCP command list. Remaining commands in this list are send using a cyclic background process. New commands are only send when the master is not waiting for a response and when response processing is finished. New sequences are inserted into the command list when starting a new sequence or during error handling. The type of processing of the newly received response message is determined using several checks (left side of the scheme).

# Commands

The XCP commands are grouped in five command groups, based on their function. Some of the commands are indicated as optional. All command definitions are added into the XCP master device driver, but only a set of STD, CAL and DAQ commands are fully implemented. Implemented commands are checked with an ‘x’ in front of the command label.

## STD - Standard commands

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Command** | **Code** | **Opt.** | **Description** |
| x | CONNECT | 0xFF | no | Set up connection with slave |
| x | DISCONNECT | 0xFE | no | Disconnect from slave |
| x | GET\_STATUS | 0xFD | no | Get current session status from slave |
| x | SYNCH | 0xFC | no | Synchronize command execution after timeout |
| x | GET\_COMM\_MODE\_INFO | 0xFB | yes | Get communication mode info |
| x | GET\_ID | 0xFA | yes | Get identification from slave |
| x | SET\_REQUEST | 0xF9 | yes | Request to save to non-volatile memory |
| x | GET\_SEED | 0xF8 | yes | Get seed for unlocking a protected resource |
| x | UNLOCK | 0xF7 | yes | Send key for unlocking a protected resource |
| x | SET\_MTA | 0xF6 | yes | Set Memory Transfer Address in slave |
| x | UPLOAD | 0xF5 | yes | Upload from slave to master |
| x | UPLOAD\_SHORT | 0xF4 | yes | Upload from slave to master (short version) |
|  | BUILD\_CHECKSUM | 0xF3 | yes | Build checksum over memory range |
|  | TRANSPORT\_LAYER\_CMD | 0xF2 | yes | Refer to transport layer specific command |
|  | USER\_CMD | 0xF1 | yes | Refer to use defined command |

## CAL - Calibration commands

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Command** | **Code** | **Opt.** | **Description** |
| x | DOWNLOAD | 0xF0 | no | Download from master to slave |
|  | DOWNLOAD\_NEXT | 0xEF | yes | Download from master to slave (block mode) |
|  | DOWNLOAD\_MAX | 0xEE | yes | Download from master to slave (fixed size) |
|  | SHORT\_DOWNLOAD | 0xED | yes | Download from master to slave (short version) |
|  | MODIFY\_BITS | 0xEC | yes | Modify bits |

## PAG - Paging commands

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Command** | **Code** | **Opt.** | **Description** |
|  | SET\_CAL\_PAGE | 0xEB | no | Set calibration page |
|  | GET\_CAL\_PAGE | 0xEA | no | Get calibration page |
|  | GET\_PAG\_PROCESSOR\_INFO | 0xE9 | yes | Get general information on PAG processor |
|  | GET\_SEGMENT\_INFO | 0xE8 | yes | Get specific information for a segment |
|  | GET\_PAGE\_INFO | 0xE7 | yes | Get specific information for a page |
|  | SET\_SEGMENT\_MODE | 0xE6 | yes | Set mode for a segment |
|  | GET\_SEGMENT\_MODE | 0xE5 | yes | Get mode for a segment |
|  | COPY\_CAL\_PAGE | 0xE4 | yes | Copy page |

## DAQ - Data acquisition and Stimulation commands

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Command** | **Code** | **Opt.** | **Description** |
| x | CLEAR\_DAQ\_LIST | 0xE3 | no | Clear DAQ list configuration |
| x | SET\_DAQ\_PTR | 0xE2 | no | Set pointer to ODT entry |
| x | WRITE\_DAQ | 0xE1 | no | Write element in ODT entry |
| x | SET\_DAQ\_LIST\_MODE | 0xE0 | no | Set mode for DAQ list |
| x | GET\_DAQ\_LIST\_MODE | 0xDF | no | Get mode from DAQ list |
| x | START\_STOP\_DAQ\_LIST | 0xDE | no | Start/stop/select DAQ list |
| x | START\_STOP\_SYNCH | 0xDD | no | Start/stop DAQ lists |
| x | GET\_DAQ\_CLOCK | 0xDC | yes | Get DAQ clock from slave |
| x | READ\_DAQ | 0xDB | yes | Read element from ODT entry |
| x | GET\_DAQ\_PROCESSOR\_INFO | 0xDA | yes | Get general information on DAQ processor |
| x | GET\_DAQ\_RESOLUTION\_INFO | 0xD9 | yes | Get general information on DAQ processing resolution |
| x | GET\_DAQ\_LIST\_INFO | 0xD8 | yes | Get specific information for a DAQ list |
| x | GET\_DAQ\_EVENT\_INFO | 0xD7 | yes | Get specific information for an event channel |
| x | FREE\_DAQ | 0xD6 | yes | Clear dynamic DAQ configuration |
| x | ALLOC\_DAQ | 0xD5 | yes | Allocate DAQ lists |
| x | ALLOC\_ODT | 0xD4 | yes | Allocate ODTs to a DAQ list |
| x | ALLOC\_ODT\_ENTRY | 0xD3 | yes | Allocate ODT entries to an ODT |

## PGM - Nonvolatile memory programming commands

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Command** | **Code** | **Opt.** | **Description** |
|  | PROGRAM\_START | 0xD2 | no | Indicate the beginning of a programming sequence |
|  | PROGRAM\_CLEAR | 0xD1 | no | Clear a part of non-volatile memory |
|  | PROGRAM | 0xD0 | no | Program a non-volatile memory segment |
|  | PROGRAM\_RESET | 0xCF | no | Indicate the end of a programming sequence |
|  | GET\_PGM\_PROCESSOR\_INFO | 0xCE | yes | Get general information on PGM processor |
|  | GET\_SECTOR\_INFO | 0xCD | yes | Get specific information for a sector |
|  | PROGRAM\_PREPARE | 0xCC | yes | Prepare non-volatile memory programming |
|  | PROGRAM\_FORMAT | 0xCB | yes | Set data format before programming |
|  | PROGRAM\_NEXT | 0xCA | yes | Program a non-volatile memory segment (block mode) |
|  | PROGRAM\_MAX | 0xC9 | yes | Program a non-volatile memory segment (fixed size) |
|  | PROGRAM\_VERIFY | 0xC8 | yes | Program verify |

# XCP information and settings

A lot of information can be requested from the slave device, in order to identify which slave is connected and what part of the XCP standard is supported by this slave. Also settings like byte order can be read in order to start communication in a way that the slave device expects.

## General XCP settings

Using the getInfo() function, all general settings are requested from the slave device. When starting a new XCP connection using the startSession() function, the getInfo() function is also executed. The information returned by this function is stored into a special XCPInfo data object. The information is gathered using multiple XCP commands: CONNECT, GET\_STATUS, GET\_COMM\_MODE\_INFO and GET\_ID. These settings are used by the master to make decisions during operation. An overview of the XCPInfo content is given in Appendix A. For more detailed information see also the specific Javadoc documentation.

Some examples of general XCP settings:

* Byte order
* Version numbers of the protocol and transport layer
* Slave identifier
* Supported resources
* Protected resources

## Data acquisition settings

Using the getDaqInfo() function, all DAQ related settings are requested from the slave device. The information returned by this function is stored into a special XCPDaqInfo data object. The information is gathered using two XCP commands: GET\_DAQ\_PROCESSOR\_INFO and GET\_DAQ\_RESOLUTION\_INFO. These settings are used by the master to make decisions during operation. An overview of the XCPDaqInfo content is given in Appendix B. For more detailed information see also the specific Javadoc documentation.

Some examples of data acquisition settings:

* Static or dynamic DAQ list configuration
* Support of configurable prescaler
* Number of available event channels
* Timestamp information

# Event channels

DAQ lists at the slave device (see next chapter) are triggered by event channels to send the values of their ODT entries. Two types of event channels are possible. Cyclic and non-cyclic event channels. Cyclic channel triggers occur at the defined sample rate of the event channel. This sample rate can be prescaled by the DAQ list in order to define custom sample rates. Non-cyclic event channels are triggered only when a defined situation happens. The event channels information available at the slave device is requested using the getEvents() function. The event channels are returned as an array of XCPEvent data objects. Each of these objects contain the parameters of its event channel, that are listed in Appendix C.

The XCP protocol offers a mechanism for receiving event information from the slave device. First the total number of available events is requested using the GET\_DAQ\_PROCESSOR\_INFO command. Then, all events information is requested for the available number of events and stored in an array of XCPEvent objects.

Some examples of event channel related settings:

* Name of the event channel
* Maximum number of DAQ lists that can use the event channel
* Whether the event channel is cyclic, if yes, what sample period is used

# DAQ lists

A DAQ list (XCPDaqList object) is a group of Object Description Tables (ODT’s, XCPOdt object) at the slave. These ODTs contain ODT entries (XCPOdtEntry object), which in their case contain a reference to an ASAP2 element available at the slave. If the slave supports, multiple DAQ lists can be configured. Each DAQ list is triggered by an individual event (see previous chapter) at the slave, with or without prescaler, and will send the values of their referenced elements to the master. The maximum number of DAQ lists, ODT’s and ODT entries might be restricted by the slave. The XCP master will return an error when trying to configure an invalid DAQ list.

There are two types of configuration modes: static and dynamic. In static mode, the number of DAQ lists, ODTs and ODT entries are already allocated. In dynamic mode this hierarchy is defined dynamically during runtime. Size and length limitations are given for both static and dynamic configurations. Also, predefined DAQ list configurations might be initiated by the slave. The current XCP implementation supports dynamic DAQ list configuration, but static configuration should be possible with minor changes.

A special DAQ list data object is defined to transfer DAQ list specific information between application layer and XCP protocol layer. That is, when receiving the currently in use DAQ list configuration, but also when configuring a new DAQ list configuration. The DAQ list object contains all information about a DAQ list configuration. A full overview of this Java class can be found in the Javadoc documentation. An overview of the XCP DAQ list specification described by ASAM can be found in the XCP protocol specification document part 1 (section 1.1.1.4). A sample of the DAQ list hierarchy can be found in Figure 9.

**XCPDaqList [0]**

**XCPOdt [0]**

|  |
| --- |
| XCPOdtEntry [0] |
| XCPOdtEntry [1] |
| XCPOdtEntry [2] |
| XCPOdtEntry [3] |
| XCPOdtEntry [4] |
| XCPOdtEntry [5] |
| XCPOdtEntry [6] |

**XCPOdtEntry [0]**

|  |
| --- |
| name |
| bit\_offset |
| address |
| address\_ext |
| size |

Figure DAQ list hierarchy

The current master only supports dynamic DAQ list configuration and DAQ direction communication mode. A DAQ list configuration is loaded into the slave device using the function setDaqLists(). When successfully configured, the DAQ lists can be started using the function toggleDaq(). In DAQ mode, each DAQ list is triggered by its specified event channel, with or without using a prescaler. A XCP DAQ message is send for each ODT. When having multiple ODT items in an ODT, it is important to check if no message overflow will occur because the number of data bytes in a message is limited, especially when a timestamp is also included. A data overflow will not generate any error, but can lead to unwanted situations because (expected) data is lost.

Incoming DAQ messages are processed by the processDaq() function. This function generates one or more XCPDaqItem objects of the received message. These objects are passed to a XCPDaqHandler object that handles the distribution of the received items within the Application Layer, using an event and listener mechanism. The XCPDaqHandler also performs central datalogging.

# Command List

The XCP functions available to the Application Layer are higher level functions, which use one or more XCP commands. These XCP commands form a sequence of commands in which in case of errors additional commands and repetitions are inserted. For controlling the sequence, a special command list object is created. This command list is an array of command list items, see Figure 10. Each command list item contains the command code, command parameters, error code and number of repeats, see Figure 11.

The XCP Protocol Layer processes this command list inside a cyclic background process at a specified sample rate. This is the same cyclic process that detects communication timeouts (see next chapter). Because the master only supports standard communication mode, each send command has first to be answered by the slave, before the next command can be send. This makes it easy to process slave responses and sending the next command. When waiting for incoming responses or during response processing, no new commands are send and only one response is expected at a time. When all commands of the list are successfully executed at the slave, the sequence is marked as completed and its status is returned.

Several command list handling functions are included in the XCPCmdList class. These are for example functions that move commands down the list, insert new or copied commands into the list or swap remaining commands back to the beginning of the command list.

The command list item information is used for command list and error handling. For more information about the XCPCmdList and XCPCmdListItem classes, see the specific Javadoc documentation.

|  |  |
| --- | --- |
| **XCPCmdList** | |
| 0 | XCPCmdListItem |
| 1 | XCPCmdListItem |
| 2 | XCPCmdListItem |
| 3 | XCPCmdListItem |
| … | XCPCmdListItem |

Figure XCPCmdList class

|  |  |
| --- | --- |
| **XCPCmdListItem** | |
| cmd | command |
| param[] | parameters |
| error | errorcode |
| repeat | no. of repeats |

Figure XCPCmdListItem class

# Error handling

When the master sends a command to the slave, no error occurs if the slave within a specified time answers with a positive response. A timeout error occurs if the slave doesn’t answer with any response within a specified time. An errorcode occurs if the slave answers within a specified time with a negative response.

## Table of error codes

|  |  |  |
| --- | --- | --- |
| **Error** | **Code** | **Description** |
| ERR\_CMD\_SYNCH | 0x00 | Command processor synchronization |
| *ERR\_TIMEOUT* | *0x01* | *Timeout* |
| *ERR\_EMPTY* | *0xFE* | *No error set* |
| *ERR\_NONE* | *0xFF* | *No error* |
| ERR\_CMD\_BUSY | 0x10 | Command was not executed |
| ERR\_DAQ\_ACTIVE | 0x11 | Command rejected because DAQ is running |
| ERR\_PGM\_ACTIVE | 0x12 | Command rejected because PGM is running |
| ERR\_CMD\_UNKNOWN | 0x20 | Unknown command or not implemented optional command |
| ERR\_CMD\_SYNTAX | 0x21 | Command syntax invalid |
| ERR\_OUT\_OF\_RANGE | 0x22 | Command syntax valid but command parameter(s) out of range |
| ERR\_WRITE\_PROTECTED | 0x23 | The memory location is write protected |
| ERR\_ACCESS\_DENIED | 0x24 | The memory location is not accessible |
| ERR\_ACCESS\_LOCKED | 0x25 | Access denied; Seed & Key is required |
| ERR\_PAGE\_NOT\_VALID | 0x26 | Selected page not available |
| ERR\_MODE\_NOT\_VALID | 0x27 | Selected page mode not available |
| ERR\_SEGMENT\_NOT\_VALID | 0x28 | Selected segment not valid |
| ERR\_SEQUENCE | 0x29 | Sequence error |
| ERR\_DAQ\_CONFIG | 0x2A | DAQ configuration not valid |
| ERR\_MEMORY\_OVERFLOW | 0x30 | Memory overflow error |
| ERR\_GENERIC | 0x31 | Generic error |
| ERR\_VERIFY | 0x32 | The slave internal program verify routine detects an error |

ERR\_TIMEOUT, ERR\_EMPTY and ERR\_NONE are custom added for internal use.

## Error handling matrix

When trying to recover from an error, the master first has to perform a pre-action and then an action. The pre-action brings the slave in a well-defined state that allows the master to perform the action.

Supported pre-actions:

* Wait t7
* SYNCH
* GET\_SEED / UNLOCK
* SET\_MTA
* SET\_DAQ\_PTR
* START\_STOP\_x
* Reinitialize DAQ

Supported actions:

* Display error
* Retry other syntax
* Retry other parameter
* Use ASAP2
* Use alternative
* Repeat 2 times
* Repeat ∞times (implemented as: Repeat 5 times)
* Restart session
* Terminate session

Timeouts are detected by the background cyclic process and are processed using the same mechanism as the other error codes.

If the master receives a negative response (error), it has to check the “error handling matrix” described by ASAM in the XCP protocol specification document part 2 (section 1.7.3.2) [3]. This matrix defines a pre-action and action for each command. For undefined situations a default pre-action and action will take place, based on the error severity. Pre-action and action are set based on received errorcode and last send command using a nested switch construction in the function processError(). Pre-action and action are then executed or added to the command list by calling the functions insertAction() and insertPreAction().

# Seed and Key mechanism

Certain groups of commands (resources) can be locked for the master by default and need to be unlocked in order to use them. The unlocking mechanism is based on requesting a seed for a specified resource from the slave, the master calculates a key from this seed using an user-defined algorithm and sends this key back to the slave to unlock the specified resource. This mechanism is described by ASAM in the XCP protocol specification document part 4 (section 2) and part 2 (section 1.6.1.2.4 and 1.6.1.2.5) [3].

## ASAM Seed and Key mechanism

As mentioned, resources can be locked at the slave device. There are three groups of commands (resources) defined by ASAM. Each resource has its own identifier and related commands.

|  |  |  |
| --- | --- | --- |
| **XCP Resource** | **Identifier** | **Commands** |
| **CAL/PAG** Calibration / Paging | 0x01 | DOWNLOAD, DOWNLOAD\_NEXT, DOWNLOAD\_MAX, SHORT\_DOWNLOAD, MODIFY\_BITS, SET\_CAL\_PAGE, GET\_CAL\_PAGE, GET\_PAG\_PROCESSOR\_INFO, GET\_SEGMENT\_INFO, GET\_PAGE\_INFO, SET\_SEGMENT\_MODE, GET\_SEGMENT\_MODE, COPY\_CAL\_PAGE |
| **DAQ & STIM**  Data Acquisition  & Stimulation | 0x04  &  0x08 | CLEAR\_DAQ\_LIST, SET\_DAQ\_PTR, WRITE\_DAQ, SET\_DAQ\_LIST\_MODE, GET\_DAQ\_LIST\_MODE, START\_STOP\_DAQ\_LIST, START\_STOP\_SYNCH, GET\_DAQ\_CLOCK, READ\_DAQ, GET\_DAQ\_PROCESSOR\_INFO, GET\_DAQ\_RESOLUTION\_INFO, GET\_DAQ\_LIST\_INFO, GET\_DAQ\_EVENT\_INFO, FREE\_DAQ, ALLOC\_DAQ, ALLOC\_ODT, ALLOC\_ODT\_ENTRY |
| **PGM**  Programming | 0x10 | PROGRAM\_START, PROGRAM\_CLEAR, PROGRAM, PROGRAM\_RESET, GET\_PGM\_PROCESSOR\_INFO, GET\_SECTOR\_INFO, PROGRAM\_PREPARE, PROGRAM\_FORMAT, PROGRAM\_NEXT, PROGRAM\_MAX, PROGRAM\_VERIFY |

Unlocking a resource requires following a defined number of steps:

* Request the seed data by sending the XCP command GET\_SEED with the correct resource identifier as parameter to the slave device.
* The slave device responds to the GET\_SEED command by sending the seed data.
* The master device driver can calculate a key from this seed, using an external function file. The interface of this external function file is described by ASAM. The external (DLL) file is normally provided by the slave device vendor. The master device calls the external method ComputeKeyFromSeed with the resource identifier and seed data as parameters. This method calculates a key using the given parameters and returns this key back to the master device driver.
* The returned key is used as a parameter for the XCP command UNLOCK. By sending this command, the slave device is asked to unlock the specified resource. The slave device responds whether the resource is successfully unlocked.

The XCP command UNLOCK returns a resource identifier bitmask that indicates which resources are protected. This bitmask can also be requested at any time using the XCP command GET\_STATUS.

## Java Native Interface

An external function file is used to calculate the key from a seed. However, the interface as described by ASAM requires that parameters of its methods are passed as pointers. Java cannot do this, so a ‘dereferenced’ interface between the external function file and Java is required. This is accomplished by writing a JNI (Java Native Interface) which is called from within the Java code. This interface then calls the external function file using pointers to variables inside the JNI. These variables inside the JNI are manipulated by the external function file, but can also be read by Java code.

The JNI is written in C++ and should be compiled to an external function file (DLL) using the following command (when using Borland C++ Builder):

bcc32 -tWD -e"SeedKeyNative.dll" XCP\_SeedKeyNative.cpp

The JNI acts as an interface for both functions provided in the external function file:

* XCP\_ComputeKeyFromSeed Computes key from seed for specified resource
* XCP\_GetAvailablePrivileges Returns bitmask of available privileges to compute a key for

More information on how this JNI is written and how it is called from within the Java code can be found in the code comments in the following files:

* JNI XCP\_SeedKeyNative.cpp
* Java XCP.SeedKeyNative.java

General information about writing a JNI can be found in the following document provided by Sun Microsystems: “The Java Native Interface - Programmer’s Guide and Specification” [7].

## Implementation

When, during operation, a command is send to the slave device that returns the ERR\_ACCESS\_LOCKED errorcode, the master automatically tries to unlock the related resource by using the seed & key mechanism and given external function file.

For each XCP command, the related resource is known. When an ERR\_ACCESS\_LOCKED errorcode is returned, the master is able to identify which resource it has to unlock, by looking to the last command that has been send to the slave device. It will request the (first part of the) seed for this resource by sending the GET\_SEED command. The slave device answers with the length and (first) data bytes of the seed. If the seed is longer than the available data bytes in a message, additional GET\_SEED commands are send for receiving all the seed data bytes. When the last part of the seed is received, the master makes a call to the JNI class SeedKeyNative to compute a key from the given seed and resource. The JNI will make a call to the external function file which computes a key and saves this in the JNI. The JNI returns whether the key has been successfully computed. When successful, the master can read the computed key out of the JNI and use it to send it by one or more UNLOCK commands. The number of required commands depends on the length of the key.

The slave device will return whether the key was valid and if the resource has been unlocked. After a successful unlock sequence, the formerly locked command is repeated and the remaining sequence is started. When the unlock sequence was not successful, the unlock sequence is repeated two more times. When still unsuccessful, an error will be returned and the currently active sequence will exit.

# Limitations

Only a part of the ASAM XCP specification is implemented in the master device driver. Not all functionality is required for basic use, or is listed for future development. An overview of present limitations is given below.

* Only XCP 1.0 implementation (not fully implemented, see chapter 7)
* Only PEAK CAN-USB transport layer implementation
* Only standard mode communication support (no block or interleaved mode communication)
* Only Motorola byte order fully supported (Intel byte order is implemented for only 80%)
* Only dynamic DAQ list configuration support (covers most of the static configuration though, but needs additional checks)
* Only ‘absolute ODT number’ Identification Field Type support
* Only byte address granularity support (others not tested)
* No XCP paging support
* No flash programming support
* No data stimulation support
* No (explicit) resume mode support
* No master-slave time synchronization
* No bypassing support
* No address calculation support

A task list and requirements for future developments to improve and extend the current XCP master driver can be found in the “HANtune overview” document [8].

# Functional tests

The current XCP master driver software has been tested for its functionality. This testing is split into two parts. First, the overall framework of command-list and message processing that is set-up has been tested for proper functioning. Second, each implemented command is tested for send information and slave response processing. All tests have been executed during development stage. Each ‘block’ of functionality is tested individually and when implemented inside the framework. An overview of the implemented commands is given in chapter 7.

Framework

* **Command list**; All methods for inserting new commands into the list, moving commands through the list and all other methods to modify the list are validated.
* **Message processing**; Each newly received message is processed by the XCP master. The type of processing is dependent on the type of message. The framework provides a mechanism to detect what type of message is received. The mechanism also detects whether the slave device response is of the format that it expects and whether it is received in a wrong sequence. This mechanism is validated.
* **Error handling**; The slave device might send an error in response back to the master device. The master has to take appropriate actions in response to the error. This error handling mechanism is validated and executes its pre-actions and actions like required by the XCP standard.
* **Timeout detection**; Timeouts may appear when the slave device sends his response too late or not at all. The timeout-length is dependent on the command that is send. The timeout detection mechanism is validated.
* **Error interface**; The interface methods of the XCP master driver do return an error code if it cannot execute properly. This mechanism is coupled to the error handling and timeout detection mechanism. Each interface method may use one or more XCP commands. Proper functioning of the error return of the currently implemented interface methods is validated.

Commands

* **Command message generation**; Each command message is an assembly of command code and parameters. The parameters come from user-input, data objects or are hardcoded into the software. Each command has its own method to assembly its command message. This mechanism is validated for each implemented command.
* **Slave response processing**; Each command that has been send expects some response of the slave device. The format of the response is known. Received data might be stored in Java variables or data objects or new actions might be initiated in response to it. Not all response processing is implemented yet, but if implemented, the processing is validated for proper functioning.

# Conclusion and recommendations

*What is the XCP protocol standard?*

XCP is an universal standardized protocol for communication between a master (tool) and slave (ECU) device, but also describes how these devices should take action in response to this communication. These actions include functionality for calibration and measurements, but also for flash programming. Several mechanisms and prescriptions are given in the standard.

*How can the XCP protocol standard be implemented in the Java programming language?*

An implementation of this standard can be made for the whole standard, but also subsets of this functionality are possible. The implementation should use the same separation between protocol and transport layer. Communication can be controlled by putting the XCP commands into a sequence. This sequence can be handled and manipulated by a cyclic background process which includes a nested decision matrix for message and error handling and functionality for sending new messages. Information can be stored into special Java data objects for internal use and exchange between the various layers. The Seed and Key mechanism can be implemented using a Java Native Interface (JNI).

Future developments should focus on improving the checking content of function parameters received from the slave device and other layers. Also, the current implementation has limited XCP support, but already provides a framework. This should be extended to a more complete XCP implementation. An overview of the future tasks and requirements is given in the “HANtune overview” documentation [8].

# References

1. Kvaser, ‘Introduction to the CCP/xCP protocols’ [web page]  
   URL: < <http://www.kvaser.com/can/misc/ccp-xcp/ccp-xcp-protocol.htm> >
2. ASAM, *CAN Calibration Protocol - Version 2.1*, 1999  
   URL: < <http://www.asam.net> >
3. ASAM, *XCP, Version 1.0, The Universal Measurement and Calibration Protocol Family*, 2003  
   URL: < <http://www.asam.net> >
4. Patzer, Andreas, *XCP at the Focal Point of Measurement and Calibration Applications*, Vector, 2007  
   URL: < <http://www.vector.com> >
5. Vector, ‘XCP’ [web page]  
   URL: < <http://www.vector.com/vi_xcp_en.html> >
6. Vector, *Calibrating ECUs optimally*, 2008  
   URL: < <http://www.vector.com> >
7. Sheng Liang, *The Java Native Interface, Programmers Guide and Specification*, Sun Microsystems, 1999  
   URL: < <http://java.sun.com/docs/books/jni/download/jni.pdf> >
8. AJ van der Hoeven, *HANtune overview*, HAN Automotive, 2010

# Appendix A - Settings in XCPinfo

Several XCP commands do return information that is stored as settings inside the XCP master driver. An overview of the settings returned by each of these commands is given. While all settings are stored, not all related implementation is done at this stage of development. All settings can be accessed as fields of the specified data object.

**Settings returned by the XCP command CONNECT:**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| support\_cal\_pag | Indicates whether Calibration and Paging support is available |
| support\_daq | Indicates whether DAQ lists support is available. |
| support\_stim | Indicates whether data stimulation mode of a DAQ list is available. |
| support\_pgm | Indicates whether Flash programming support is available. |
| byte\_order | Indicates the byte order as used by the XCP slave device. |
| address\_granularity | Indicates the address granularity. |
| slave\_block\_mode | Indicates whether the XCP slave device supports block communication mode from slave to master. |
| comm\_info\_optional | Indicates whether the command GET\_COMM\_MODE\_INFO is supported. |
| protocol\_version | Indicates the XCP protocol layer version used by the XCP slave device. |
| transport\_version | Indicates the XCP transport layer version used by the XCP slave device. |

**Settings returned by the XCP command GET\_STATUS:**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| store\_cal\_req | Indicates whether STORE\_CAL\_REQ is set. |
| store\_daq\_req | Indicates whether STORE\_DAQ\_REQ is set. |
| clear\_daq\_req | Indicates whether CLEAR\_DAQ\_REQ is set. |
| daq\_running | Indicates whether data transfer is running. |
| slave\_resume | Indicates whether the slave is in RESUME mode. |
| protected\_cal\_pag | Indicates whether the Calibration and Paging commands are protected. |
| protected\_daq | Indicates whether the DAQ list commands are protected. |
| protected\_stim | Indicates whether the data stimulation commands are protected. |
| protected\_pgm | Indicates whether the Flash programming commands are protected. |
| session\_id | Session configuration identifier. |

**Settings returned by the XCP command GET\_COMM\_MODE\_INFO:**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| master\_block\_mode | Indicates whether the XCP slave device supports block communication mode from master to slave. |
| interleaved\_mode | Indicates whether the interleaved communication mode is supported. |
| xcp\_driver\_version | Indicates the version number of the XCP driver used in the slave device. |

**Settings returned by the XCP command GET\_ID:**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| slave\_identifier | Array of slave identifier bytes. |
| slave\_identifier\_len | Indicates the length of the slave identifier array. |

**XCP Limits:**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| MAX\_CTO | Indicates the maximum CTO packet size in bytes. |
| MAX\_DTO | Indicates the maximum DTO packet size in bytes. |
| MAX\_BS | Indicates the maximum allowed block size as the number of consecutive command packets. |
| MAX\_ST | Indicates the minimum required separation time between the packets in block communication mode in units of 100 microseconds. |
| QUEUE\_SIZE | Indicates the maximum number of consecutive packets in interleaved communication mode the master can send to the receipt queue of the slave device. |
| OS | Indicates the offset at which the command codes start. |

# Appendix B - Settings in DAQinfo

Several XCP commands do return information that is stored as settings inside the XCP master driver. An overview of the settings returned by each of these commands is given. While all settings are stored, not all related implementation is done at this stage of development. All settings can be accessed as fields of the specified data object.

**Settings returned by the XCP command GET\_DAQ\_PROCESSOR\_INFO:**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| daq\_config\_type | Indicates whether the DAQ configuration is static or dynamic. |
| prescaler\_supported | Indicates whether DAQ lists can prescale an event channel. |
| resume\_supported | Indicates whether DAQ lists can be added to a RESUME mode configuration. |
| bit\_stim\_supported | Indicates whether bitwise data stimulation is supported. |
| timestamp\_supported | Indicates whether timestamp mode is supported. |
| pid\_off\_supported | Indicates whether the Identification Field can be turned off. |
| daq\_overload | Indicates the overload identification type. |
| daq\_id\_type | Indicates the Identication Field type. |
| daq\_optimisation\_type | Indicates the Optimisation Type. |
| daq\_address\_ext | Indicates the Address Extension type. |
| min\_daq | Indicates the number of predefined DAQ lists. |
| max\_daq | Indicates the total number of DAQ lists available at the slave device. |
| max\_event\_channel | Indicates the maximum number of event channels available at the slave device. |

**Settings returned by the XCP command GET\_DAQ\_RESOLUTION\_INFO:**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| daq\_odt\_entry\_granularity | Indicates the granularity for size of an ODT entry for DAQ direction. |
| daq\_odt\_entry\_size\_max | Indicates the maximum size of an ODT entry for DAQ direction. |
| stim\_odt\_entry\_granularity | Indicates the granularity for size of an ODT entry for STIM direction. |
| stim\_odt\_entry\_size\_max | Indicates the maximum size of an ODT entry for STIM direction. |
| timestamp\_mode | Indicates the timestamp unit and size. |
| timestamp\_ticks | Indicates the timestamp ticks per unit. |

# Appendix C - Settings for events

Several XCP commands do return information that is stored as settings inside the XCP master driver. An overview of the settings returned by each of these commands is given. While all settings are stored, not all related implementation is done at this stage of development. All settings can be accessed as fields of the specified data object.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| daq\_allowed | Indicates whether DAQ direction is allowed for this event. |
| stim\_allowed | Indicates whether STIM direction is allowed for this event. |
| max\_daq\_list | Indicates the maximum number of DAQ lists that can use this event. |
| name | Indicates the name of this event channel. |
| time\_cycle | Indicates whether this event is cyclic and what sampling period is used by the slave for this event. |
| time\_unit | Indicates the timestamp unit that is used by the slave for defining the time unit for this event. |
| priority | Indicates the priority of this event. |