(https://www.kvaser.com/)

CCP/XCP

Background

The CCP (CAN Calibration Protocol) is, just as the name indicates, a protocol for calibration of and data acquisition from electronic control units (ECU). The protocol is defined by **ASAM (Association for Standarisation of Automation- and Measuring Systems) (http://www.asam.net/)**, earlier known as ASAP (Arbeitskreis zur Standardisierung von Applikationssystemen). This is an international organization consisting of a number of significant vehicle manufacturers i.e. Audi, BMW, VW etc. Until now different technical solutions have been used for developing, calibration, production and service of ECU hardware and software. The aim with CCP is to create a common tool for all stages of ECU developing and which is compatible with different kinds of hardware and software.

The ASAM group defines a lot of standards. The CCP and XCP standards are found in subsection AE (Automotive Electronics) and are grouped into something called **AE MCD 1**. The current version of the CCP specification is 2.1, released in February, 1999.

Areas of Use

The most common area of use is in the automotive industry, where CAN is often used, but also in other industry where CAN is used. Conceivable areas of use include –

- ECU developing.
- Systems for function and environmental tests of ECUs.
- In a test bench for combustion engines, gear box or climate control.
- For measurements and calibration in pre-production vehicles.
- In a general CAN application outside vehicle industry.

Quick facts

CCP is a application layer for CAN 2.0B (11- or 29-bits CAN id). The Protocol is a top layer (layer 7) according to the OSI model, which means that the protocol does not describe how bits and bytes are created but uses the CAN 2.0B protocol physical, data link and network layer.

CCP supports the following functions:

- Reads and writes to ECU memory.

- Synchronous cyclic data acquisition from a ECU.
- Simultaneous calibration and data acquisition.
- Handles multiple nods on the CAN bus.
- Flash programming.
- Plug and play.
- Protection of resources (data acquisition and calibration).

(https://www.kvaser.com/software_application/calibration/#?applications=calibration)

Find ECU Calibration Software

(https://www.kvaser.com/software_application/calibration/#?applications=calibration)

Find CCP/XCP software applications for ECU Calibration.

Use the “Request More Information” button to get in touch with the Technical Associate.

(https://www.kvaser.com/software_application/calibration/#?applications=calibration)

(https://www.kvaser.com/software_application/data-acquisition/)

Find Data Acquisition Software

(https://www.kvaser.com/software_application/data-acquisition/)

Find CCP/XCP software applications for Data Acquisition.

Use the “Request More Information” button to get in touch directly with the Technical Associate.

(https://www.kvaser.com/software_application/data-acquisition/)

In Depth

The CCP in detail

CCP is built on a master/slave application where CCP-master starts communication by sending commands to a slave node. There can be several slave nodes connected on a CAN bus. CCP uses generic commands for data acquisition and simple memory handling for calibration. These two resources are independent and can therefore be used simultaneously.

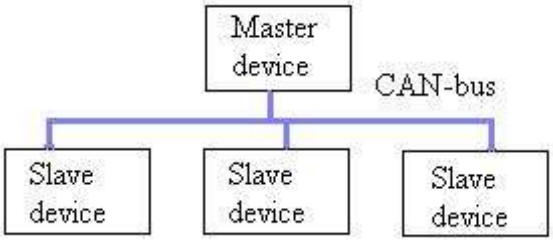

Figure 1: CCP bus connection

CCP has been designed to handle the restrictions and demands of both small 8-bits microcontrollers and ECUs with high performance. No extra hardware has to be connected to the ECU. The CCP driver is fully implemented in the software. A simple implementation of CCP only needs a small part of the available RAM, ROM and CPU time for execution. A simple implementation only need two CAN message identifiers, which can be set as low priority that does not disturb the ordinary traffic. If CCP is to be used from an ordinary PC, the same simple and low cost CAN interface which is used in microcontrollers can be used.

Generic commands

CCP uses generic commands, which are not node specific, to perform different functions in a slave node. As the commands are generic every node must have an individual station address. A logical connection between master and slave has to be set up before any commands can be sent. This connection persists until the master decides to connect to another slave node or until the master sends a disconnect command. After the connection the master controls all communication between master and salve. Every message from the master is followed by a reply message from the slave containing data or error codes.

CCP-specific CAN messages

CCP is built on the CAN 2.0B protocol. All messages are 8 bytes long.

Only two types of CAN messages is needed, CRO and DTO, one for each direction. The CRO (Command Recieive Object) messages are sent from master to slave and contain control commands. DTO (Data Transmission Object) messages are sent from slave to master. When a slave has received a CRO message it performs the given instructions and then answers with a DTO message containing a CRM (Command Return Message). The CRM code tells the master if the corresponding control command has been performed as planned or not.

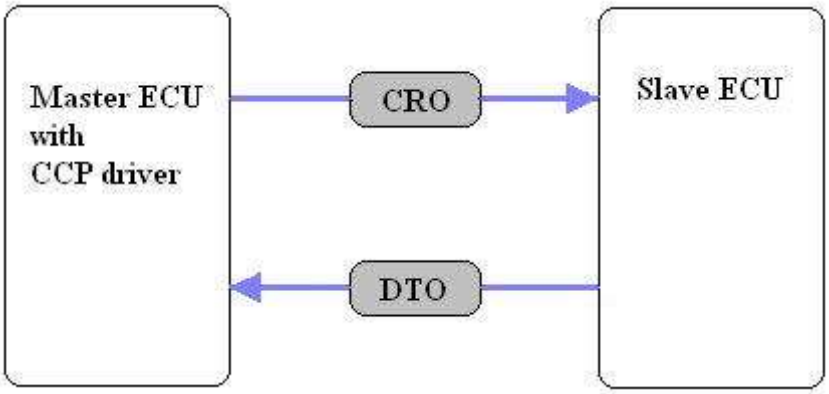

Figure 2: CRO and DTO messages.

The CAN identifiers used for the CRO and DTO messages are determined by a configuration file ("A2L file", defined by the **ASAM MCD 2MC/ASAP2** standard) which is used to configure the master. The configuration file may also contain information about the slave memory organization, which is useful for data acquisition and calibration. Since the CAN identifiers define the priority of the message it should be chosen in a way that does not disturb the ordinary traffic on the bus. CCP does not determine which byte order to use (Motorola or Intel) in general data transmission. There is an exception that states that the byte order for the station address used when establishing a connection between master and slave has to be Intel (LSB first).

Description of CRO messages

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
CMD	CTR	Data	Data	Data	Data	Data	Data

Figure 3: Organization of CRO message.

CRO messages are sent from master to slave and contain instructions. The first byte is a command code (CMD) which describes the purpose of the message. The second byte is a command counter (CTR) and is used for keeping track of the communication. The command counter is also expected to be sent in return in the DTO message from the slave. Bytes 2-7 are reserved for data parameters depending on the command code. A message is always 8 bytes long and bytes which are not defined are considered as do not care".

Description of DTO messages

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
CMD	ERR	CTR	Data	Data	Data	Data	Data

Figure 4: Organization of DTO message.

The DTO message is sent by the slave as a receipt of a received CRO message and it is also used for data acquisition. The first byte in the message is called PID (Packet ID). The value of the PID describes the message type. There are three types of messages:

- 0xFF, command return message (CRM), if the DTO is sent as a receipt of a CRO message.
- 0xFE, event message, if the DTO reports internal slave status changes in order to invoke error recovery or other services.

- 0 – 0xFD, data acquisition message (DAQ). This PID has the value of an ODT (Object Descriptor Table), which is described later.

Data Acquisition (DAQ)

The master device can initiate and start data acquisition from the slave device. Data is sent from the slave with special DAQ-DTOs. The data bytes are organized in DAQ lists which consists of a number of ODT lists. An ODT list contains up to 7 pointers to memory addresses in the ECU where data is stored. Besides pointers to memory addresses a ODT-list can contain a address extension and the number of bytes to be sent. All slave devices don't handle data elements longer than one byte and it is up to the master to solve this task by splitting up the data into single bytes.

DAQ-DTO consists of a PID and the data elements of which the memory pointers in the ODT list points at. The PID number (usually the same as the ODT list) has a value between 0 and 253 which means that there can be only 254 ODT lists simultaneously.

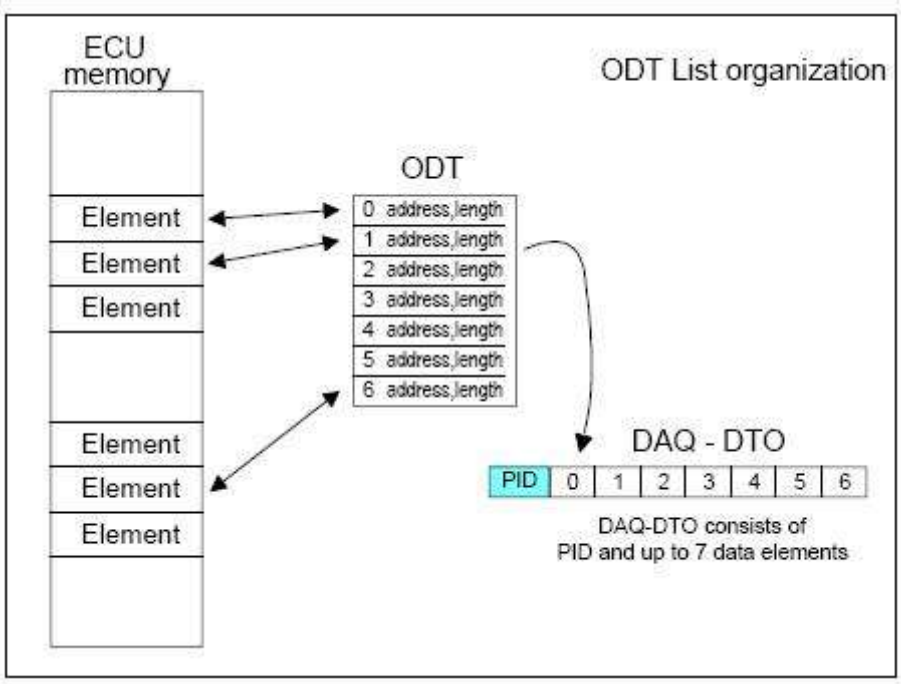

Figure 5: Organization of ODT list.

The CCP specification allows several DAQ lists, which can be simultaneously active. Transmission of DAQ lists is initiated by the master through a START_STOP command. Data bytes in the ODT lists are sampled in the slave device and then sent on the CAN bus in DAQ-DTOs. If a new START_STOP command is received by the slave before the ongoing DAQ cycle is finished, there are two ways to react. The new DAQ command is started and the ongoing is terminated or the ongoing cycle is finished and the new is ignored. There are advantages and disadvantages with both methods and the CCP specification doesn't say which one to choose.

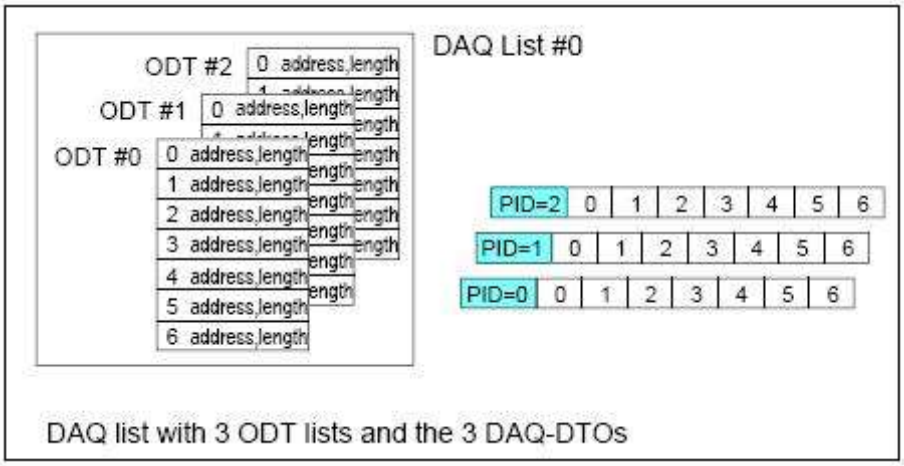

Figure 6: Organization of DAQ lists.

Commands

The following commands are included in the CCP specification (Figure 7). All commands don't have to be implemented if the ability for calibration isn't needed and are therefore marked as optional in the table below (Figure 7). Also GET_DAQ_SIZE, SET_DAQ_PTR, WRITE_DAQ, START_STOP (and START_STOP_ALL) are DAQ specific and don't have to be implemented unless this resource is used. GET_SEED and UNLOCK are used to unlock CCP resources like data acquisition and calibration if they are protected by a key (password), which is optional.

Command	Code	TimeOut to ACK [ms]	Remark
CONNECT	0x01	25	
GET_CCP_VERSION	0x1B	25	
EXCHANGE_ID	0x17	25	
GET_SEED	0x12	25	optional command
UNLOCK	0x13	25	optional command
SET_MTA	0x02	25	
DNLOAD	0x03	25	
DNLOAD_6	0x23	25	optional command
UPLOAD	0x04	25	
SHORT_UP	0x0F	25	optional command
SELECT_CAL_PAGE	0x11	25	optional command
GET_DAQ_SIZE	0x14	25	
SET_DAQ_PTR	0x15	25	
WRITE_DAQ	0x16	25	
START_STOP	0x06	25	
DISCONNECT	0x07	25	
SET_S_STATUS	0x0C	25	optional command
GET_S_STATUS	0x0D	25	optional command
BUILD_CHKSUM	0x0E	30 000	optional command
CLEAR_MEMORY	0x10	30 000	optional command
PROGRAM	0x18	100	optional command
PROGRAM_6	0x22	100	optional command
MOVE	0x19	30 000	optional command
TEST	0x05	25	optional command
GET_ACTIVE_CAL_PAGE	0x09	25	optional command
START_STOP_ALL	0x08	25	optional command
DIAG_SERVICE	0x20	500	optional command
ACTION_SERVICE	0x21	5 000	optional command

Figure 7: Commands.

Error Handling

Depending on the error code from the slave and how critical it is, the master take different actions that are described in Figure 8.

Error C0 is a warning and no action is taken. If error C1 occurs there is an error in the communication or the node sending it is busy. On error C1 the master should wait the ACK time, described in figure 7, and then try to resend the message. This should be done two times. Error C2 might be a temporary power loss and can be solved by a re-initialization. Error C3 is irresolvable and the master should terminate the running session.

“Cold Start” means that a new logical connection between master and slave is established with a CONNECT command and some further initialization.

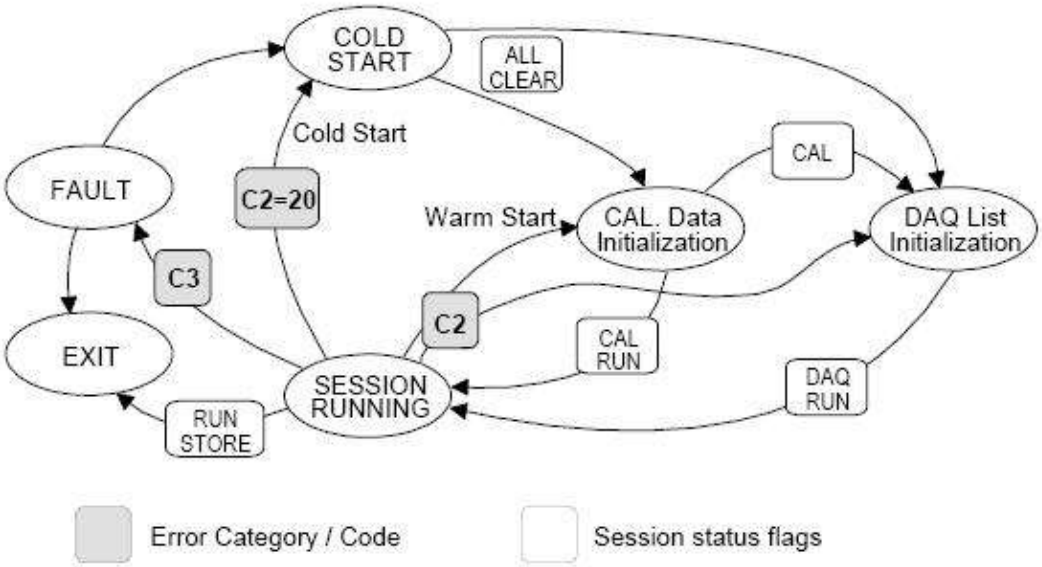

Figure 8: Error handling.

Example of Sequences

The examples describe commands to use, as master, for basic CCP communication.

Login session (Cold Start)

A typical connection between master and slave starts with a CRO containing a CONNECT command from the master. The slave should answer with a corresponding DTO. To make sure that both master and slave talks the same “language” a GET_CCP_VERSION command is sent from the master with the expected version number. If the version number sent in return from the slave matches, communication can proceed. For “plug ‘n play” compatible nodes EXCHANGE_ID command can be used for automatic session configuration, depending on station address. On command GET_SEED the slave node returns information about protection status (locked/unlocked) of resources (DAQ or Calibration). If, for some reason, a resource is locked it has to be unlocked before it can be used. To unlock

a resource an UNLOCK command, with a “key” received in the GET_SEED DTO, has to be sent from the master. Before the login session is ended initialization of status bits are recommended, this is done with SET_S_STATUS command.

Calibration init session

This session description assumes that a login procedure has been performed. The first thing then would be to set the session status bit for calibration to “off” with SET_S_STATUS command. Thereafter the memory address containing the data to exchange is selected with SET_MTA command. To ensure that this memory address is available a BUILD_CHKSUM command is sent and an answer from the slave node that confirms this is expected. Thereafter the data byte(s) can be downloaded to the selected address. First a DOWNLOAD command is sent with the number of data bytes and the value of each byte. To actually perform the data exchange the SELECT_CAL_PAGE is sent. To indicate that calibration has started session status bit for calibration is set to “on” with SET_S_STATUS command.

DAQ init session

This session description assumes that a login procedure has been performed. The session status bit for DAQ is set to “off” with SET_S_STATUS. A DAQ list is selected with GET_DAQ_SIZE and the slave answers with the number of available ODTs. SET_DAQ_POINTER command selects which DAQ list, ODT table and element in ODT that should be written to. WRITE_DAQ command then assigns the memory address of the data parameter to the previous selected element. When all DAQ lists are filled as wanted the session status bit for DAQ is set to “on” using SET_S_STATUS. The transmission of a DAQ list is started with the START_STOP command. If several DAQ lists should be started at the same time and sent synchronously START_STOP_ALL command is used.

Higher Layer Protocols
(https://www.kvaser.com/about-can/higher-layer-protocols/)

PHONE

+46 31 886344 (tel:+46 31 886344)

EMAIL

sales@kvaser.com (mailto:sales@kvaser.com)