CCP Driver

Implementation in Electronic Control Units

CCP Version 2.1

Vector Informatik GmbH, Ingersheimer Strasse 24, 70499 Stuttgart
Tel. 0711/80670-0, Fax 0711/80670-111, Email can@vector-informatik.de
Internet http://www.vector-informatik.de

Contents

1 Introduction

1.1 Reading order of CCP related documents

This document should be used as a reference manual for the Vector CCP Driver version 2.1. For understanding the concepts of CCP, we propose to start your reading with the following documents:

➢ Ccpintro.pdf	General overview of CCP
➢ Ccp21.pdf	Official CCP 2.1 specification
➢ CCP_Test.pdf	Application note, which explains in detail how to integrate the Vector CCP driver into an ECU with an existing CAN driver

1.2 Overview

The CAN Calibration Protocol (CCP) was standardized by the European ASAP working committee for standardization of interfaces used in calibration and measurement data acquisition. CCP is a higher level CAN protocol used for communication between a measurement and calibration system (MCS, i.e. CANape) and an electronic control unit (ECU).

This document describes the implementation and customization of the Vector CCP driver in an ECU. The Vector CCP driver is provided as ANSI C source code, which is linked together with the ECU's application program code. A simple application program interfaces the CCP driver to the CAN driver and to the ECU application program. The functionality of the Vector CCP driver may be customized to meet the requirements of the application.

The source code of the CCP driver and documentation is available on the Internet at www.vector-informatik.de.

1.3 File Description

Name	Type	Format	Description
`CCP.C`	C-Source	Text	Contains the CCP driver source code.
`CCP.H`	C-Header	Text	Definition of the interface between driver and ECU and some constants.
`CCPPAR.H`	C-Header	Text	#define statements for customizing the driver.
`CAN_CCP.C`	C-Source	Text	Interface to the Vector CAN driver.

1.4 Change History

Author	Versi on	Date	Changes
Zaiser	1.00	12/02/98	Created
Zaiser	1.10	01/08/99	CCP Version V2.1
Zaiser	1.12	02/15/99	Minor changes (ccpSendCallBack)
Zaiser	1.16	03/19/99	Description of Vector driver interface. Interrupt considerations
Zaiser	1.18	04/19/99	CAN driver handshake example. Pointer definitions.
Zaiser	1.29	09/24/00	Review V20 removed
Hänger	1.31	03/05/01	Expanded interface description
Zaiser	1.34	01/07/02	CCP Driver Version 1.34
Deckardt	1.37	17/07/2002	CCP Driver Version 1.37

2 Interface Description

2.1 General

The CCP driver needs a CAN driver to transmit and receive CAN messages. The Vector CCP driver does not include a CAN driver. The CCP driver communicates with the CAN driver via a simple application program interface (see section 2.2).

Usually, only two CAN messages are needed for the communication between the MCS and the ECU via CCP. One CAN message for transmitting and one CAN message for receiving:

CCP Designation	Direction
CRO	MCS ⇒ ECU
DTO	ECU ⇒ MSC

The CCP driver contains two process units:

- ➢ a command processor, here the communication between the MCS and the ECU follows a strict query-reply sequence. The CRO message contains commands, e.g. read memory or write memory, which are then executed by the CCP driver. Each command is acknowledged by a DTO message.
- ➢ a data acquisition (DAQ) processor, which is used for automatic cyclic or event driven data acquisition. Here the communication between the MCS and the ECU is done in one direction, DTO messages are transmit from the ECU to the MCS.

The task of the CAN driver in the ECU is to call CCP functions from the CCP driver when receiving a CRO or after transmitting a DTO message. The communication parameters (identifier, baud rate, etc.) of the ECU are defined by the initialization function of the used CAN driver.

The CCP driver utilizes the function ccpSend() to transmit a reply or a data message. After the message has been transmitted successfully, the CAN driver has to call the function ccpSendCallBack() in the CCP driver to indicate the successful transmission.

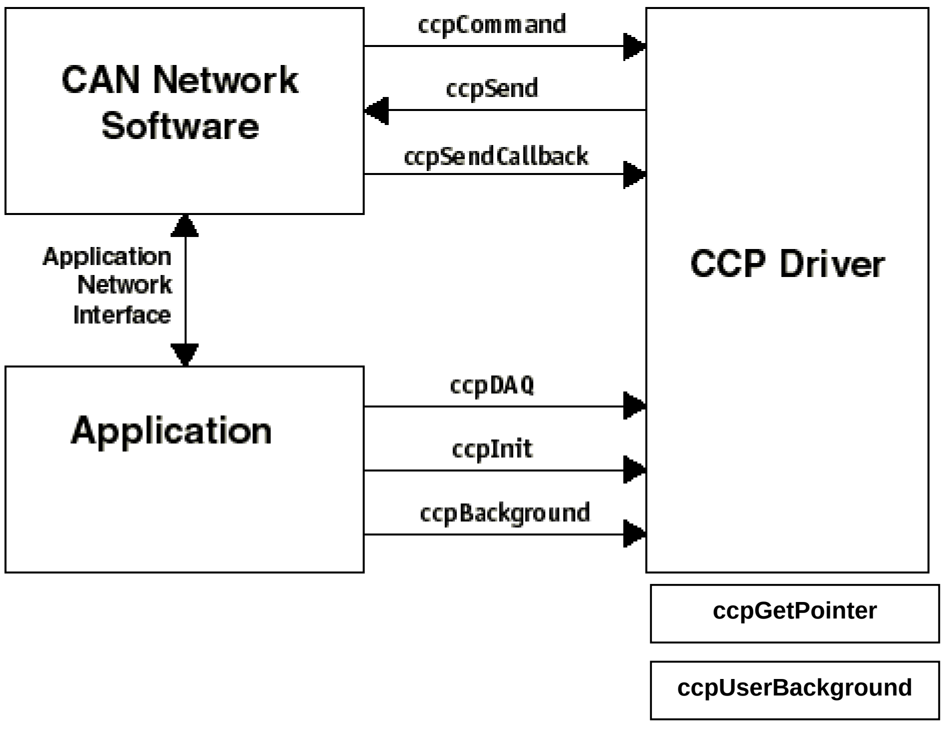
If a CRO message is received, the CAN driver calls the command processor function ccpCommand() of the CCP driver.

2.2 Functional Interface Overview Diagram

The interface of the CCP driver is based on function calls. The functions ccpInit, ccpDaq and ccpBackground are called from the application program. The functions ccpCommand and ccpSendCallback are called from the CAN driver. The CCP driver calls ccpSend which calls a transmitting function of the CAN driver. The ccpGetPointer is a function, which is called inside by the CCP driver. The ccpUserBackground function is called from the ccpBackground function. For many implementations the function body is empty.

Note: ccpDaq and ccpBackground are only needed in some instances.

The functions ccpCommand, ccpSend, ccpSendCallback, ccpGetPointer and ccpUserBackground are called or defined in file can_ccp.c (see page 4).



See also section 2, “Overview” from the CCP_Test.pdf document.

The following C pseudo code example shows the required software handshake between the CCP driver and the CAN driver:

```
/* Main Loop */
for (;;) {

  /* CRO Message received */
  if (CROReceiveBufferFull) {
    ccpCommand(data);
    CROReceiveBufferRelease = true;
  }

  /* Transmit Message Buffer available */
  if (DTOTransmitBufferEmpty) {
    ccpSendCallBack();
  }

  /* Background Processing */
  ccpBackground();
}

/* Transmit Function */

void ccpSend( CCP_BYTEPTR *data ) {
  DTOTransmitBuffer = data;
  DTOTransmitRequest = true;
  return 1;
}
```

See also section 2.4, “Integration of the Vector CCP Driver into an ECU” from the CCP_Test.pdf document.

2.3 Description of Interface Functions

2.3.1 Function ccpInit

Task	Basic initialization of the CCP driver.
Function prototype	`void ccpInit( void )`
Description	This function must be called from the application program (e.g. after the initialization of the CAN driver) before any other CCP function is called.

2.3.2 Function ccpCommand

Task	Evaluation of the CRO message, command interpreter.
Function prototype	`void ccpCommand( CCP_BYTE *cmd )`
Description	Every time the CAN driver receives a CCP CRO message, this function has to be called (e.g. in the interrupt receive routine of the CAN driver).

2.3.3 Function ccpSendCallBack

Task	Indicates that ccpSend may be called again.
Function prototype	`CCP_BYTE ccpSendCallBack( void )`
Description	The CCP driver will not call ccpSend again, until ccpSendCallBack has indicated the successful transmission of the previous message. ccpSendCallBack will try to transmit any pending DTO messages. ccpSendCallBack returns 0 (FALSE), when the CCP driver is idle (no transmit messages where pending).

2.3.4 Function ccpSend

Task	Transmit request of a DTO CAN message to the CAN driver
Function prototype	`void ccpSend(CCP_BYTEPTR msg )`
Description	Requests for transmission of the DTO message. ccpSendCallBack() has to be called after successfull transmission of the DTO message. The CCP driver will not request further transmissions, until ccpSendCallBack() has been called. The CAN identifier shall be CCP_DTO_ID, which is defined in CCPPAR.H.

2.3.5 Function ccpDaq

This is an optional function. This function has only to be called from the application program if the data acquisition mode is enabled (see page 14). See also section 2.6, “Enabling the Synchronous Data Acquisition Mode” from the CCP_Test.pdf document.

Task	Sample measurement data for the given event channel and start to transmit the data via DTO CAN messages to the MCS. Optional CCP command.
Function Prototype	`void ccpDaq(CCP_BYTE eventChannel )`

Description	Version 2.1 of CCP supports up to 256 different event channels. Calling ccpDaq() with a particular event channel number, triggers the sampling and transmission of all DAQ lists that are assigned to this event channel.
	
	The event channels are defined by the ECU developer in the application program. An MCS (e.g. CANape) must know about the meaning of the event channel numbers. These are usually described in the tool configuration files or in the interface specific part of the ASAP2 database.
	
	A motor control unit for example, may have a 10ms, a 100ms and a crank synchronous event channel. In this case, the three ccpDaq calls have to be placed at the appropriate locations in the ECU’s program:
	
	`ccpDaq(1); /* 10ms cycle */`
	`ccpDaq(2); /* 100ms cycle */`
	`ccpDaq(3); /* Crank synchronous cycle */`

2.3.6 Function ccpGetPointer

Task	Pointer conversion.

Function Prototype	`CCP_MTABYTEPTR ccpGetPointer(CCP_BYTE ext,`
	`                              CCP_DWORD addr )`

Description	This function converts a memory address from CCP format (32-bit address plus 8-bit address extension) to a C style pointer. An MCS like CANape usually reads this memory addresses from the ASAP2 database or from a linker map file.
	
	The address extension may be used to distinguish different address spaces or memory types. In most cases, the address extension is not used and may be ignored.
	
	A typical implementation of ccpGetPointer may look like this:
	
	`CCP_MTABYTEPTR ccpGetPointer( CCP_BYTE ext,`
	`                              CCP_DWORD ddr ) {`
	`  return (void near *)addr;`

	}

2.3.7 Function ccpBackground

This is an optional function. This function has only to be called from the application program if the checksum calculation mode is enabled – see page 21 - or if CCP commands should not be executed on interrupt level – see page 31. See also section 2.7, “Checksum Calculation” from the CCP_Test.pdf document.

Task	Performs memory checksum calculations in the background. Optional CCP command.
Function Prototype	`CCP_BYTE ccpBackground( void )`
Description	If the CCP command for the calculation of the memory checksum has to be done over large memory areas, then it is not appropriate to block the processor for a long period of time. Therefore, the checksum calculation is splitted into small chunks that are handled in ccpBackground. For this reason ccpBackground should be called periodically whenever the ECU’s CPU is idle. ccpBackground returns 1 (TRUE), if the checksum calculation is still in progress.

2.3.8 Function ccpUserBackground

Task	Performs additional, user specific calculations.
Function Prototype	`void ccpUserBackground( void )`
Description	This function is only called, if the ccpBackground function is also called from the application program. Additional functions can be called here. In most cases the function body is empty.

2.4 Configuration of Basic Functionality

Configuration of basic functionality is done in file ccppar.h (see chapter 1.3). See also section 2.2, “Configuring the CCP Driver” from the CCP_Test.pdf document.

If you use a CAN driver from Vector, then you can configure the Vector CCP driver via the CANgen tool.

2.4.1 CAN Identifiers for CRO and DTO

Defines the CAN identifiers for the CRO and the DTO messages. Both CAN ID’s must be set appropriately in the MCS.

Example:

```
#define CCP_CRO_ID 636
#define CCP_DTO_ID 637
```

2.4.2 CCP Station address

If multiple ECUs with CCP are on the bus with unique CRO identifiers, the station address serves to identify a particular ECU when establishing the connection. The station address must be set appropriately in the MCS.

Example:

```
#define CCP_STATION_ADDR   0x0039
```

2.4.3 CCP Station Identifier

The CCP station identifier is an ASCII string that identifies the ECU’s software program version.

The MCS can interpret this identifier as file name for the ECU database. The ECU developer should change the CCP station identifier with each program change. This will prevent database mixups and grant the correct access of measurement and calibration objects from the MCS to the ECU. Another benefit of the usage of the CCP station identifier is the automatic assignment of the correct ECU database at program start of the MCS via the Plug&Play mechanism. The Plug&Play mechanism prevent the user to choose the wrong ECU database.

Example:

```
#define CCP_STATION_ID “V980101a”
```

2.4.4 Enable and Disable Interrupts

The functions ccpDaq(), ccpSendCallBack(), ccpBackground() and ccpCommand() are not reentrant. If one of these function may interrupt one of the others, the macros CCP_DISABLE_INTERRUPT and CCP_ENABLE_INTERRUPT have to be defined to protect critical sections in the code from beeing interrupted. The CCP driver will not nest the sections with disabled interrupt. The time periods are as short as possible, but note that ccpSend() will be called with disabled interrupts!

Example:

```
#define CCP_ENABLE_INTERRUPT asm „SETIE“
#define CCP_DISABLE_INTERRUPT asm „CLRIE“
```

3 Optional Features of CCP Driver

The Vector CCP driver offers some optional features. You can configure these additional CCP options with "defines" in the ccppar.h file. If you use a CAN driver from Vector you can configure your CCP driver using the CANgen tool. For some optional CCP features, additional functions must be provided by the ECU developer. These functions are implementation specific.

3.1 Data Acquisition Mode (DAQ)

See also section 2.6, “Enabling the Synchronous Data Acquisition Mode” from the CCP_Test.pdf document.

3.1.1 General Description

Description	Defining CCP_DAQ will enable the table driven data acquisition mode. In this mode, the MCS configures tables of memory addresses in the CCP driver. These tables contains pointers to measurement objects, which have been configured previously for the measurement in the MCS. Each configured table is assigned to an event channel. The ECU developer has to add *ccpDaq(x)* function calls in his ECU source code at all the places where the ECU has to transmit the measurement signals. The ECU outputs automatically the current value of the measurement objects via DTO messages to the MCS, when the function ccpDaq() is executed in the ECU’s code with the corresponding event channel number. This means that the data can be transmitted at any particular point of the ECU code when the data are valid. During the measurement the MCS does not send any command to the ECU. This leads to less bus load. There are some other defines which makes only sense if they are used together with the CCP_DAQ define. These defines configure different options for the data acquisition mode.
Defines	CCP_DAQ
Implementation Specific Functions	-
CANape settings	Open the dialog CCP Device Setup with the menu command Options|Driver parameters. Go to tab Event. Make one entry for each event channel. An event channel is an ccpDaq(x) function call in ECU source code.
CCP compliance	Yes.

ECU code	Function ccpDaq() must be called each time with an different event channel number.

3.1.2 Configuration of the tables

Description	It is possible to have several DAQ lists (as many as CCP_MAX_DAQ). Each DAQ list may be assigned to a particular event channel and can contain multiple ODTs (as many as DAQ_MAX_ODT). An ODT describes the contents of a single CAN message for measurement data acquisition. Each ODT stores the address information for the output of 7 data bytes (1 byte is used for differentiation) in the CAN message.
	
	Each DAQ list has a supplemental RAM requirement of 4 bytes.
	
	An event channel and a prescaler is assigned to the DAQ list, i.e. the number of DAQ lists determines the number of different event channels and sampling rates that are simultaneously possible.
	
	The number of data bytes that can be measured with a DAQ list corresponds to the value of CCP_MAX_ODT*7.
	
	Consequently, the overall size of the configuration table and the maximum number of output data can be determined according to the following formulas:
	
	`Memory requirement [Bytes] =`
	`      {sizeof(void*) * (7 * CCP_MAX_ODT)} * CCP_MAX_DAQ + CCP_MAX_DAQ*6`
	
	`Output data [Bytes] =`
	`      (7 * CCP_MAX_ODT * CCP_MAX_DAQ)`
	
	If the memory space requirement permits, it makes sense to use as many DAQ lists as possible with exactly one ODT each.

Defines	CCP_MAX_ODT x CCP_MAX_DAQ x
Implementation Specific Functions	-
CANape settings	-
CCP compliance	Yes.

3.1.3 Send Queue

Most times DAQ lists are used together with a send queue. If you don't want to use a send queue, your function ccpSend() must be able to handle multiple messages or you must enable the send single mode, which is described later.

Description	The option CCP_SEND_QUEUE enables the transmit queue for data acquisition messages (DTOs) in the CCP driver. The size of the transmit queue is determined by CCP_SEND_QUEUE_SIZE. The appropriate value of CCP_SEND_QUEUE_SIZE depends on the maximum number of DTOs that may be sent in one cycle (minimum: CCP_MAX_DAQ*CCP_MAX_DTO) and the bus latency time for the messages. The latency time depends on the bus load situation, the priority of the DTO message and the burst probability on the CAN bus. The best way to determine the optimum value for CCP_SEND_QUEUE_SIZE is by using the ECU on a real bus load situation and increasing the value for CCP_SEND_QUEUE_SIZE until no more messages are lost. The result may be a compromise between message loss probability and RAM needed for the queue.
Defines	CCP_SEND_QUEUE CCP_SEND_QUEUE_SIZE x
Implementation Specific Functions	-
CANape settings	-
CCP compliance	Yes.

3.1.4 Send Single Mode

Description	This option forces the CCP driver to send only a single DAQ-DTO message per ccpDaq() call. If more than one ODT are defined, the CCP driver will alternate the transmission of the corresponding DAQ-DTO messages.
Defines	CCP_SEND_SINGLE
Implementation Specific Functions	-
CANape settings	-
CCP compliance	Yes.

3.1.5 Send Queue Overrun

This is an optional setting for DAQ lists.

Description	The CCP driver will drop data acquisition cycles, if the transmission queue overflows. CCP_SEND_QUEUE_OVERRUN_INDICATION enables automatic detection of overflow situations. CANape will display the number of lost cycles in the status bar. To avoid overflows, the size of the transmission queue may be increased or the number of data acquisition channels must be reduced.
Defines	CCP_SEND_QUEUE_OVERRUN_INDICATION
Implementation Specific Functions	-
CANape settings	Open the dialog CCP Device Setup with the menu command Options|Driver parameters. Go to tab CCP Parameters. Select Overload detection.
CCP compliance	No.

3.1.6 RAM Consumption Reduction

This is an optional setting for DAQ lists.

Description	Use this option, if your ECU uses 32 bit addresses for measurement objects and all measurement objects which should be measured with DAQ lists lie within a 64k space. The define CCP_DAQ_BASE_ADDR stands for the lowest address.
Defines	CCP_DAQ_BASE_ADDR x
Implementation Specific Functions	ccpGetDaqPointer()
CANape settings	-
CCP compliance	Yes.

3.1.7 Function ccpGetDaqPointer

Task	Task of this function is similar to the task of ccpGetPointer. See page 10. In difference the return value is reduced with the amount of the define CCP_DAQ_BASE_ADDR.
Function Prototype	`CCP_DAQBYTEPTR ccpGetPointer(CCP_BYTE ext, CCP_DWORD addr)`

3.2 Seed and Key

3.2.1 General Description

Description	Seed and Key allows individual access protection for calibration, FLASH programming and data acquisition with DAQ lists. The ECU sends during the connection login a seed (a few data bytes) to CANape. CANape has now to calculate a key with the help of a special algorithm and to send the calculated key back to the ECU. The calculation of the key is done with the received seed and a DLL named SEEDKEY1.DLL, which is developed by the ECU manufacturer and which has to be located in the EXEC directory of CANape. CANape can access the ECU only if the ECU has accepted the key. If the key is not valid, the ECU is locked.
Defines	CCP_SEED_KEY
Implementation Specific Functions	ccpGetSeed ccpUnlock
CANape Settings	Open the dialog CCP Device Setup with the menu command Options|Driver parameters. Go to tab CCP Parameters. Select Seed+Key.
CCP compliance	Yes.

3.2.2 Function ccpGetSeed

Task	Detect seed
Function Prototype	`CCP_DWORD ccpGetSeed( CCP_BYTE resourceMask )`
Description	There are three defines for resource masks: PL_CAL, PL_PGM, PL_DAQ. This function returns the seed. `#define CCP_SEED_KEY` in CCPPAR.H, if it is required.

3.2.3 Function ccpUnlock

Task	Check key and unlock protection
Function Prototype	`CCP_BYTE ccpUnlock( CCP_BYTE *key )`
Description	key is a Pointer to the key. It returns 0 if paramater contains a bad key. For good keys it returns one of the following defines PL_CAL, PL_PGM, PL_DAQ. `#define CCP_SEED_KEY` in CCPPAR.H, if it is required.

3.2.4 Example Implementation for SEEDKEY1.DLL

The function call of ASAP1A_CCP_ComputeKeyFromSeed() is standardized by the ASAP committee.

Datei SEEDKEY1.H

```
#ifndef _SEEDKEY_H_
#define _SEEDKEY_H_

#ifndef DllImport
#define DllImport  __declspec(dllimport)
#endif

#ifndef DllExport
#define DllExport  __declspec(dllexport)
#endif

#ifdef SEEDKEYAPI_IMPL
#define SEEDKEYAPI DllExport __cdecl
#else
#define SEEDKEYAPI DllImport __cdecl
#endif

#ifdef __cplusplus
extern "C" {
#endif

BOOL SEEDKEYAPI ASAP1A_CCP_ComputeKeyFromSeed(BYTE *seed, unsigned short sizeSeed,
BYTE *key, unsigned short maxSizeKey, unsigned short *sizeKey);

#ifdef __cplusplus
}
#endif
#endif
```

Datei SEEDKEY1.C

```
#include <windows.h>
#define SEEDKEYAPI_IMPL
#include "SeedKey1.h"

extern "C" {

BOOL SEEDKEYAPI ASAP1A_CCP_ComputeKeyFromSeed(BYTE *seed, unsigned short sizeSeed,
BYTE *key, unsigned short maxSizeKey, unsigned short *sizeKey)
{
  // in that example sizeSeed == 4 is expected only
  if( sizeSeed != 4 ) return FALSE;
  if( maxSizeKey < 4 ) return FALSE;
  *((unsigned long*)key) *= 3;
  *((unsigned long*)key) &= 0x55555555;
  *((unsigned long*)key) *= 5;
  *sizeKey = 4;
  return TRUE;
}
}
```

3.3 Checksum Calculation

See also section 2.7, “Checksum Calculation” from the CCP_Test.pdf document.

3.3.1 General Description

Description	Checksum or CRC calculation is enabled with the CCP_CHECKSUM define. The ccpBackground function has to be called. See page 11.
Defines	CCP_CHECKSUM
Implementation Specific Functions	-
CANape settings	Open the dialog CCP Device Setup with the menu command Options|Driver parameters. Go to tab CCP Parameters. Activate checksum.
CCP compliance	Yes.

3.3.2 Checksum Calculation

You can use either 3.3.2 Checksum Calculation or 3.3.3. CRC Calculation.

Description	Use define CCP_CHECKSUM_TYP to enable the checksum calculation and to choose the checksum type.
Defines	CCP_CHECKSUM_TYP BYTE or CCP_CHECKSUM_TYP WORD or CCP_CHECKSUM_TYP DWORD
Implementation Specific Functions	-
CANape settings	Open the dialog CCP Device Setup with the menu command Options|Driver parameters. Go to tab CCP Parameters. Set the corresponding calculation algorithm.
CCP compliance	Yes.

3.3.3 CRC Calculation

You can use either 3.3.2 Checksum Calculation or 3.3.3. CRC Calculation.

Description	Use define CCP_CHECKSUM_CCITT to enable CRC calculation. Define CCP_MOTOROLA or CCP_INTEL to specify the byte order of your target. CCITT the datatype is always WORD.
Defines	CCP_CHECKSUM_CCITT CCP_MOTOROLA or CCP_CHECKSUM_CCITT CCP_INTEL
Implementation Specific Functions	-
CANape settings	Open the dialog CCP Device Setup with the menu command Options|Driver parameters. Go to tab CCP Parameter. Select the corresponding calculation algorithm from the selection list.
CCP compliance	Yes.

3.3.4 Block Size

A default value (256 bytes) is used as block size for the checksum calculation. Use this option to change the default value of the block size.

Description	The checksum calculation will not be done by ccpCommand(), because of runtime considerations. The checksum calculation will be split into blocks of CCP_CHECKSUM_BLOCKSIZE size and then calculated at each ccpBackground() call.
Defines	CCP_CHECKSUM_BLOCKSIZE
Implementation Specific Functions	-
CANape settings	-
CCP compliance	Yes.

3.4 EEPROM Access

3.4.1 Reading Access

Description	For uploading data from the ECU the CCP commands CCP_UPLOAD and CCP_SHORT_UPLOAD are used. This option allows EEPROM access for these commands. If the memory address set with the CCP_SET_MTA or the CCP_SHORT_UPLOAD command lies within the EEPROM memory, then the memory is accessed. In the other case RAM is accessed.
Defines	CCP_READ_EEPROM
Implementation Specific Functions	CcpCheckReadEEPROM
CCP compliance	Yes.

3.4.2 Function ccpCheckReadEEPROM

Task	Optional reading access to EEPROM.
Function Prototype	`CCP_BYTE ccpCheckReadEEPROM( CCP_MTABYTEPTR addr, CCP_BYTE size,CCP_BYTEPTR data )`
Description	Function ccpCheckReadEEPROM() detects whether the address lies withhin the EEPROM memory. If not, the function returns with 0. If the address lies within the EEPROM memory, reading is performed and the function returns with 1. `#define CCP_READ_EEPROM` in CCPPAR.H, if required.

3.4.3 Writing Access

Description	For downloading CCP commands, the CCP_DNLOAD and CCP_DNLOAD6 commands are used. This option allows EEPROM access for these commands. If the memory address set with the CCP_SET_MTA command lies within the EEPROM memory, then the memory is accessed. In the other case RAM is accessed.
Defines	CCP_WRITE_EEPROM
Implementation Specific Functions	ccpCheckWriteEEPROM ccpCheckPendingEEPROM
CCP compliance	Yes.

3.4.4 Function ccpCheckWriteEEPROM

Task	Optional writing access to EEPROM.
Function Prototype	`CCP_BYTE ccpCheckWriteEEPROM( CCP_MTABYTEPTR addr,CCP_BYTE size,CCP_BYTEPTR data )`
Description	The function ccpCheckWriteEEPROM detects whether the address lies withhin the EEPROM memory. If not, the functions returns with CCP_WRITE_DENIED. If it lies within, EEPROM programming is performed. The function may return during programming with CCP_WRITE_PENDING or may wait until the programming sequence has finished and then returns with CCP_WRITE_OK. If the function ccpCheckWriteEEPROM returns during the programming sequence, ccpCheckPendingEEPROM must check state of the EEPROM. See next item. If the programming sequence has finished, the ccpSendCrm function must be called. CcpSendCrm is an internal function of the CCP driver. `#define CCP_WRITE_EEPROM` in CCPPAR.H, if required.

3.4.5 Function ccpCheckPendingEEPROM

Task	Check wheater EEPROM writing has finished.
Function Prototype	`void ccpCheckPendingEEPROM ( void )`
Description	The function ccpCheckPendingEEPROM is only needed if the function ccpCheckWriteEEPROM returns before the EEPROM programming has finished. See item before. The function must be called inside the function ccpUserBackground. The function ccpBackground must also be called cyclic in your application. When the EEPROM progamming has finished, ccpCheckPendingEEPROM must call ccpSendCrm. ccpSendCrm is an internal CCP function. In all the other cases ccpCheckPendingEEPROM does nothing. `#define CCP_WRITE_EEPROM` in CCPPAR.H, if required.

3.5 Flash Programing

There are two basic possibilities to program the flash memory of the ECU:

3.5.1 Flash Programing with CCP commands

Description	This method uses the CCP commands CCP_CLEAR_MEMORY CCP_PROGRAM and CCP_PROGRAM6. The functions ccpFlashClear and ccpFlashProgram must contain implementation specific code to clear and to program the flash memory. The function ccpSetCalPage is required for the parameter flash programming feature of CANape. This function disables or enables the read access of the calibration parameter RAM . ccpSetCallPage may also be used for switching between using the RAM or the FLASH parameters in the ECU.
Defines	CCP_PROGRAM
Implementation Specific Functions	ccpFlashClear ccpFlashProgramm ccpSetCalPage – see page 27.
CANape settings	Open the dialog CCP Device Setup with the menu command Options|Driver parameters. Go to tab FLASH. Select the entry “Direct” in the Flashkernel selection list.
CCP compliance	Yes.

3.5.2 Function ccpFlashClear

Task	Clear flash sector.
Function Prototype	`void ccpFlashClear(CCP_MTABYTEPTR a, CCP_ DWORD size)`
Description	The task of this function is to clear the flash memory before the flash memory will be reprogrammed. `#define CCP_PROGRAM` in CCPPAR.H, if required.

3.5.3 Function ccpProgramm

Task	Flash programming.
Function Prototype	`CCP_BYTE ccpFlashProgramm(CCP_BYTEPTR data, CCP_MTABYTEPTR a, CCP_BYTE size )`
Description	This function performs the flash programming. `#define CCP_PROGRAM` in CCPPAR.H, if required.

3.5.4 Flash Programing with a Flashkernel

Description	Using a Flashkernel is a new basic approach to program the flash memory of the ECU. The flash kernel will be downloaded to the ECU's RAM by CANape and then executed. The flash kernel contains a little CCP driver, a CAN driver and the flash programming functions. Thus a flashkernel has to be customized to the used microcontroller and to the used memory type. Flashkernels for different controllers are included in CANape. If You want write a flash kernel by yourself,You can request a flashkernel application note at Vector.
Defines	CCP_BOOTLOADER _DOWNLOAD
Implementation Specific Functions	ccpDisableNormalOperation
CANape settings	Open the dialog CCP Device Setup with the menu command Options|Driver parameters. Go to tab FLASH. Select in the Flashkernel selection list the corresponding fkl file for the used microcontroller.
CCP compliance	No. It is an extension of Vector.

3.5.5 Function ccpDisableNormalOperation

Task	This function checks whether it is possible to stop the application program and whether the memory area which is described in parameter list is free for downloading of a flashkernel. If one or both conditions are not true, the function returns 0.
Function Prototype	`CCP_BYTE ccpDisableNormalOperation(CCP_MTABYTEPTR a, CCP_WORD size )`
Description	This optional function provides FLASH programming. `#define CCP_BOOTLOADER_DOWNLOAD`

3.6 Calibration Memory

There are two possibilities to access the calibration memory, via RAM and via flash memory. Note: according the CCP protocol there is no need to configure FLASH and RAM memory. If no configuration is done, the memory access is done into the RAM.

3.6.1 Page Switching

Description	CANape is able to switch between a flash page and a RAM page. The CCP command SET_CAL_PAGE is used to activate the wished page. For flash memory programming the calibration parameter addresses are redirected to the flash memory. Redirection is done with functions ccpGetPointer and ccpProgramm.
Defines	CCP_CALPAGE
Implementation Specific Functions	ccpSetCalPage ccpGetCalPage
CANape settings	Open the dialog CCP Device Setup with the menu command Options|Driver parameters. Go to tab FLASH. Activate page switching. Enter a flash selector value (e.g. 1) and a RAM selector value (e.g. 0).
CCP compliance	Yes.

3.6.2 Function ccpGetCalPage

Task	Returns the start address of the current active calibration page.
Function Prototype	`CCP_DWORD ccpGetCalPage( void )`
Description	`#define CCP_CALPAGE` in CCPPAR.H, if required.

3.6.3 Function ccpSetCalPage

Task	Set the active calibration page.
Function Prototype	`Void ccpSetCalPage( CCP_DWORD a )`
Description	`#define CCP_CALPAGE` in CCPPAR.H, if required.

3.6.4 Calibration RAM Offset

Description	To distinguish between the flash memory and the RAM memory, an offset need to be declared. There is no additonal need for the ECU's program. Redirection of calibration parameter addresses is done by the application tool.
Defines	-
Implementation Specific Functions	-
CANape settings	Open the dialog CCP Device Setup with the menu command Options|Driver parameters. Go to tab FLASH. Fill out CALRAM offset.
CCP compliance	Yes.

3.6.5 ECU Timestamps

Description	The time stamps of the underlaying CAN protocol are used for CCP. This means, that all time stamps refers to the CAN bus time. The time precision is in general high enough for nearly all applications, except for the monitoring of ECU internal events like the monitoring of the OSEK operating system. In such cases, the monitoring with heigher precision based on the ECU's time is recommended.
Defines	CCP_TIMESTAMPING
Implementation Specific Functions	ccpGetTimestamp ccpClearTimestamp
CANape settings	Open the dialog CCP Device Setup with the menu command Options|Driver parameters. Go to tab Other. Select Use ECU time stamps. Fill in the value for Ticks/ms.
CCP compliance	No.

3.6.6 Function ccpClearTimestamp

Task	Function is called on measurement start and resets the time stamp to 0.
Function Prototype	`void ccpClearTimestamp( void )`
Description	`#define CCP CCP_TIMESTAMPING` in CCPPAR.H, if required.

3.6.7 Function ccpGetTimestamp

Task	Returns the current time stamp.
Function Prototype	`CCP_WORD ccpGetTimestamp( void )`
Description	`#define CCP CCP_TIMESTAMPING` in CCPPAR.H, if required.

3.7 Write Access Protection

3.7.1 General Description

Description	Check if access to the given memory location is allowed.
Defines	CCP_WRITE_PROTECTION
Implementation Specific Functions	ccpCheckWriteAccess
CCP compliance	Yes.

3.7.2 Function ccpCheckWriteAccess

Task	Optional memory write protection.
Function Prototype	`CCP_BYTE ccpCheckWriteAccess(CCP_BYTE *a, CCP_BYTE n )`
Description	This optional function provides write protection for particular memory addresses. It has to return 0, if the given memory address range is write protected. `#define CCP_WRITE_PROTECTION` in CCPPAR.H, if required.

3.8 Execute CCP Commands not on interrupt level

Description	The function ccpCommand is called in the notification function for the CRO object. Depending on the CAN driver this may happen on interrupt level. The runtime of ccpCommand() is usually short, but this may not be the case if EEPROM programming is required. With this feature the command handling can be disabled on interrupt level. The interface to the CAN driver is described on page 6 The function ccpCommand is described on page 9.
Defines	CCP_CMD_NOT_IN_INTERUPT
Implementation Specific Functions	-
CANape settings	-
CCP compliance	Yes.

4 Example Implementations

There are several sample implementations available:

4.1 Example Implementation for the Infineon C167

4.2 Example Implementation for the Motorola HC12

See the application note “CCP_Test.pdf“, which explains in detail how to integrate the Vector CCP driver into an HC12 microcontroller with an existing CAN driver.

5 Resource Requirements

The CCP driver requires ROM, RAM, and computing time in the ECU to fulfill its function. Based on our experience a ROM requirement of 1-2 KByte can be assumed. The RAM requirement is very dependent upon the maximum number of possible measurement channels. Current implementations lie in the range of 50-100 Bytes. The RAM requirement of the implementation described here is given in a later chapter. The computing time requirement for an existing implementation on a Siemens C167 running at 20MHz is approx. 100-200µs per measurement cycle and about 50-200µs for command execution.

6 Data Types

To provide platform independant code, there are #defines for the basic datatypes.

```
Example:


#define CCP_INTEL
#define CCP_MOTOROLA


#define CCP_BYTE unsigned char
#define CCP_WORD unsigned short
#define CCP_DWORD unsigned long
#define CCP_BYTEPTR unsigned char *
```

Some 16 bit microcontroller offer 16 bit near pointers and 32 bit far pointers. Using far pointers for data acquisition will nearly double the amount of memory needed for the DAQ lists. Therfore the type of pointers used for memory access and for data acquisition can be defined individually:

```
Example:


#define CCP_MTABYTEPTR huge unsigned char *
#define CCP_DAQBYTEPTR near unsigned char *
```

7 Limitations

The limitations of this particular CCP implementation are listed below:

- All DAQ lists contain an equal number of ODTs (CCP_MAX_ODT).
- Dynamic or static assignment of different CAN identifiers to DAQ lists is not supported.
- For performance reasons, the response to the CCP commands DNLOAD and UPLOAD does not contain the incremented MTA value.
- The implemented checksum algorithm is a BYTE, WORD or DWORD sum over all bytes in the given memory range or a WORD CCITT algorithm.
- The resume feature is not implemented.

8 Abbreviations

ASAP	Working Committee for the Standardization of MCS
ASAM	Association for Standardisation of Automation and Measuring
CAN	Controller Area Network
CCP	CAN Calibration Protocol
CRO	Command Receive Object
DTO	Data Transmission Object
DAQ	Data Acquisition Object
ODT	Object Descriptor Table
PID	Packet Identifier

9 Reference

1. **CCP CAN Calibration Protocol**
   ASAP Standard
   Version 2.1
   June 1998

2. **CCP, A CAN Protocol for Calibration and Measurement Data Acquisition**
   Rainer Zaiser
   Vector Informatik GmbH
   Ingersheimer Str. 24
   70499 Stuttgart,Germany

3. **CANape Manual V3.1**
   Vector Informatik GmbH
   Ingersheimer Str. 24
   70499 Stuttgart,Germany

4. **ISO/DIS 11898**
   Road vehicles - Interchange of digital information
   Controller Area Network (CAN) for high speed communication
   August 1992

5. **ISO/DIS 11519-1**
   Road Vehicles - Interchange of digital Information
   Low speed serial data communication,
   Part 1: Low speed Controller Area Network (CAN)

6. **CAN-Driver Vector**
   Documentation
   Volker Ebner
   Version 1.18 6.8.98

7. **Application note “CCP_Test“**
   Marco Konrad, Marcel Iannizzi
   Vector Informatik GmbH
   Ingersheimer Str. 24
   70499 Stuttgart,Germany
   September 2001