Application note:

Integration of the

Vector CCP Driver with Vector CAN Driver

▪ CCP DEMO ▪

on a Motorola HC12 microcontroller

CARD12.D60 Evaluation Board

Author:	Marco Konrad		
Version:	1.1.1		
Status:	in progress (in progress / completed / verified / released)		
Customer:			
Printing date:	15.10.01	Number of pages:	52

This application note describes how to integrate the Vector CCP Driver into an ECU and how to interface it with the Vector Can Driver.

Revision history

Version	Editor	Description
1.1.1	Ko	completed part “Explaining DAQ”
1.1	Ko	Seed and Key, details changed
1.0	Ko	Documentation started

1 Table of Contents

2 Overview

This overview summaries in general how to integrate the Vector CCP Driver into an ECU and demonstrates how to interface it with an existing VECTOR CAN Driver. Detailed information about integrating the CCP Driver step by step into a Motorola HC12 microcontroller is described in later chapters with the CCP_Demo sample.

2.1 VECTOR CAN and CCP Driver

The integration of the Vector CCP Driver requires an existing ECU with the Vector CAN Driver. To enable the CCP functions, the Vector CAN Driver has to be configured with the CANGen tool (**see section 2.2.2**), which comes up with the delivery. For a short impression about the associations between the CANGen tool an the CAN/CCP Driver configurations and the sngle project files have a look at the following pictures.

2.1.1 Configuration of the Vector CAN/CCP Driver

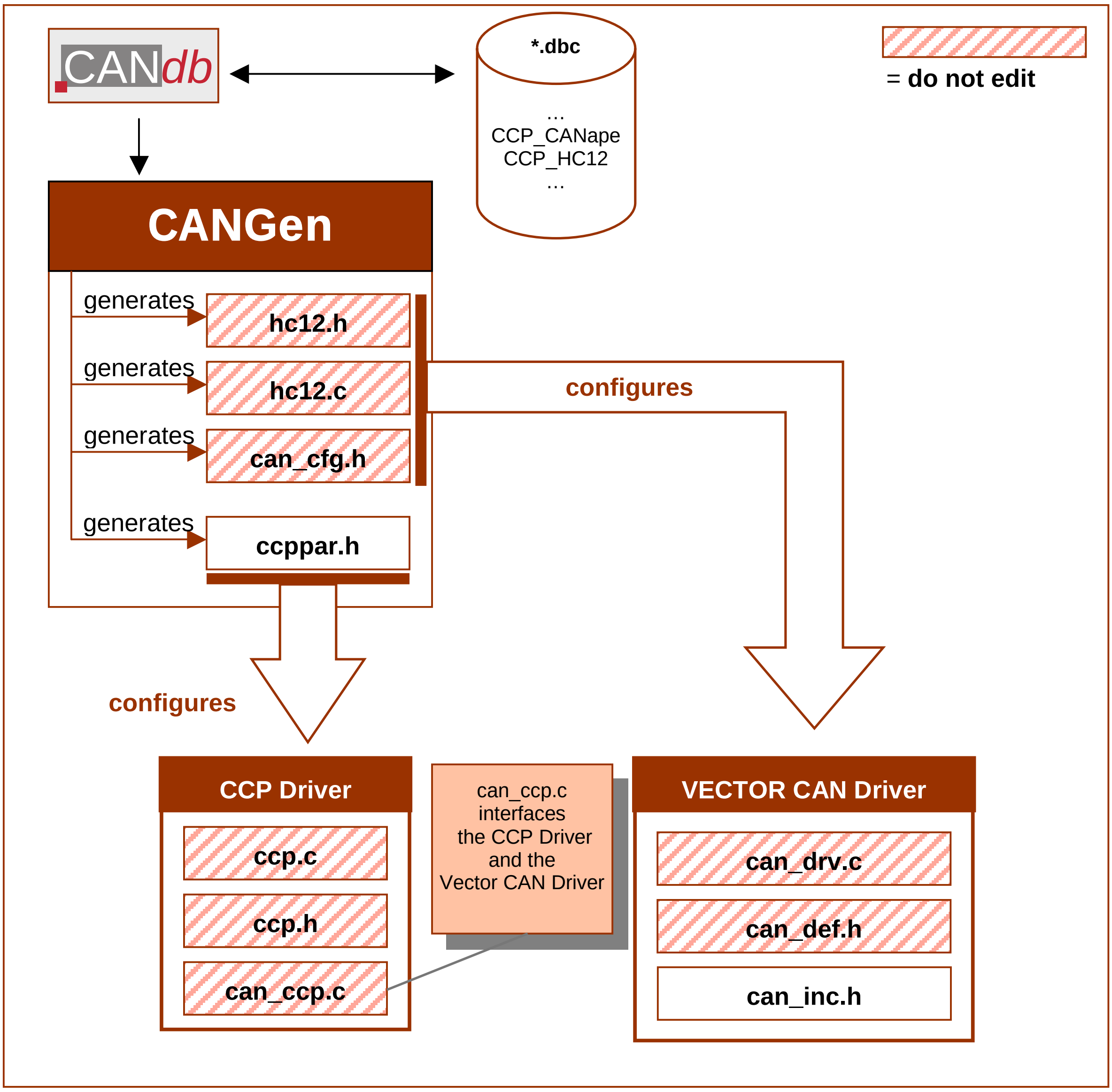


Figure 1: Configuration of Vector CAN/CCP Driver

2.1.2 Project Files of User's Application

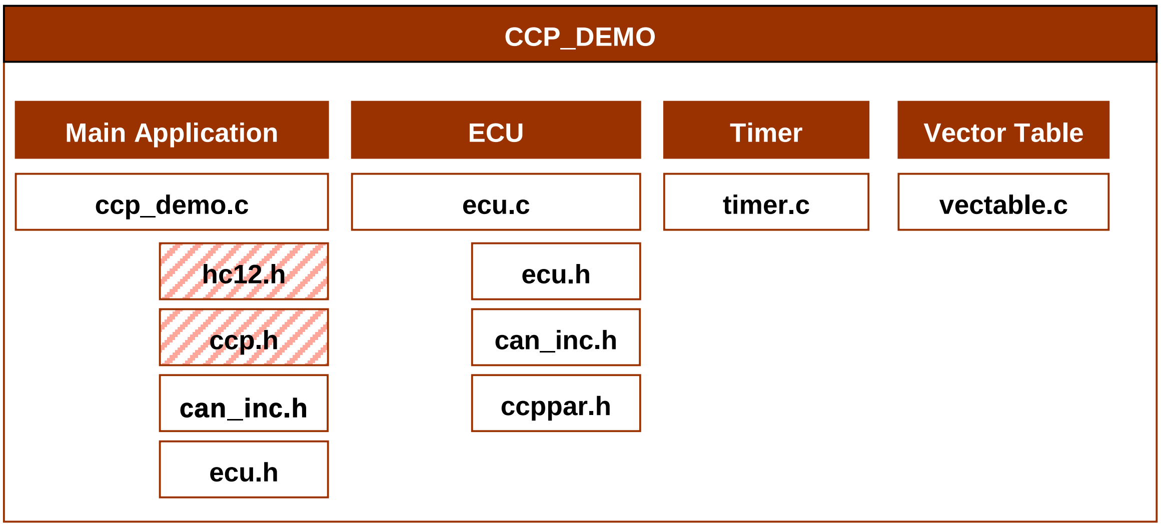


Figure 2: source file integration in CCP_DEMO sample

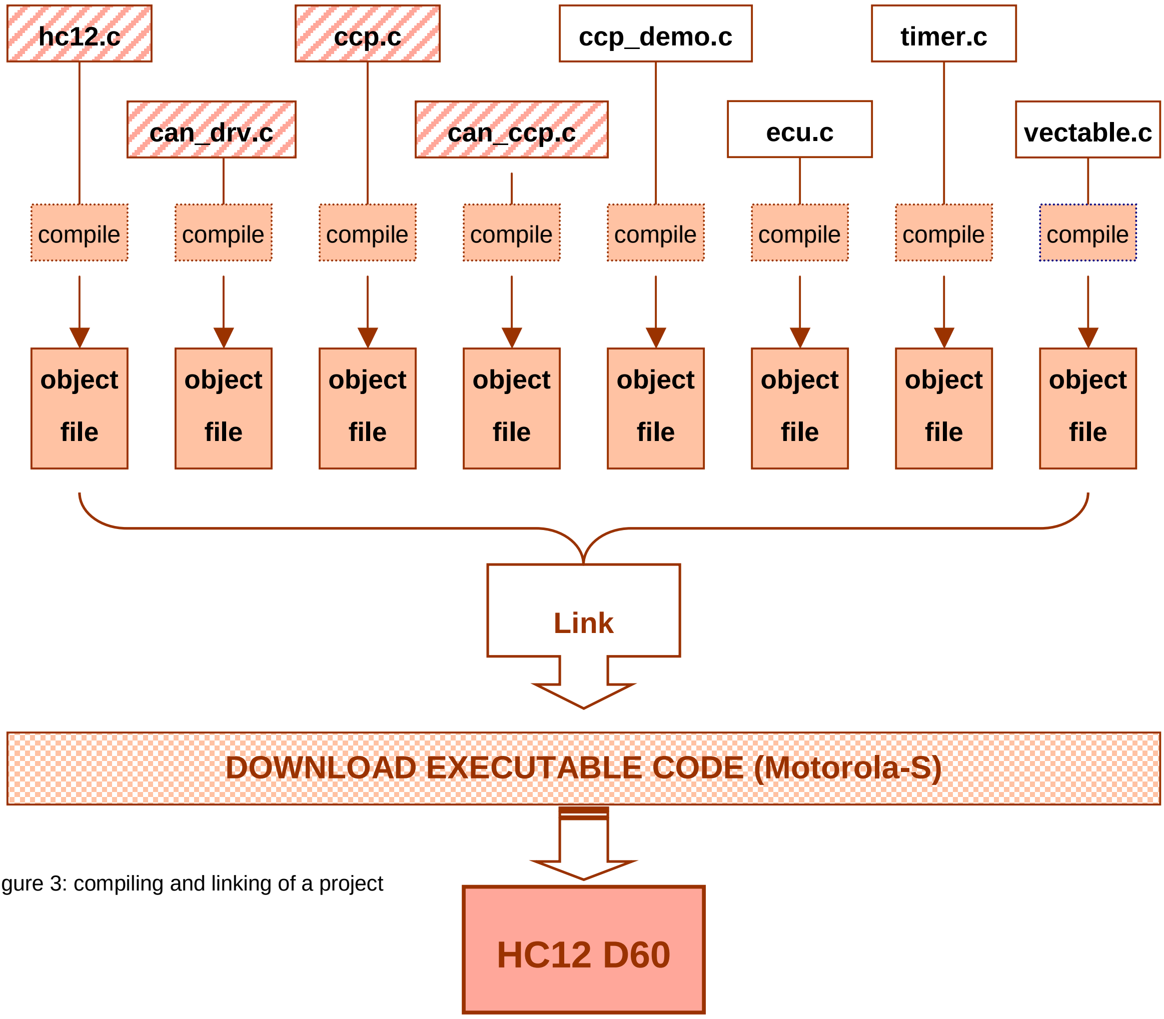


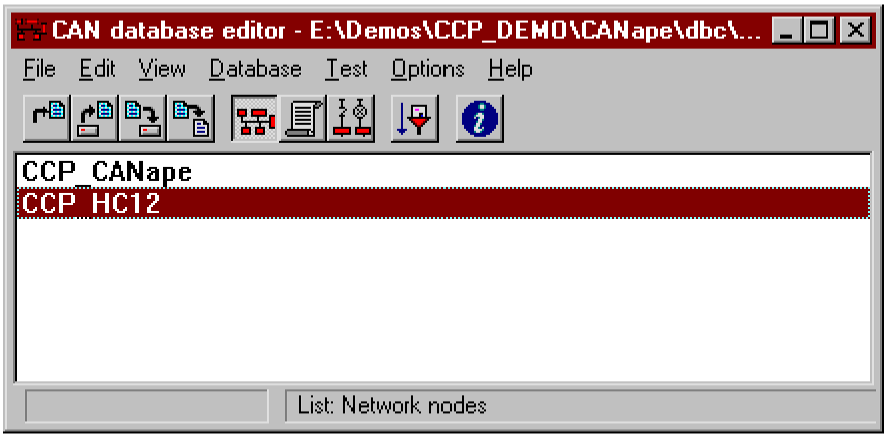
Figure 3: compiling and linking of a project

2.2 Tools

2.2.1 The Database Editor CAN*db*

The Database Editor is needed to create a new *.dbc file or to add new messages into an existing one.
To set up the CCP Driver for the Vector CAN Drive, the user has to add two new nodes and two new messages in an existing database, which is used by the **CANGen** tool (see section 2.2.2).

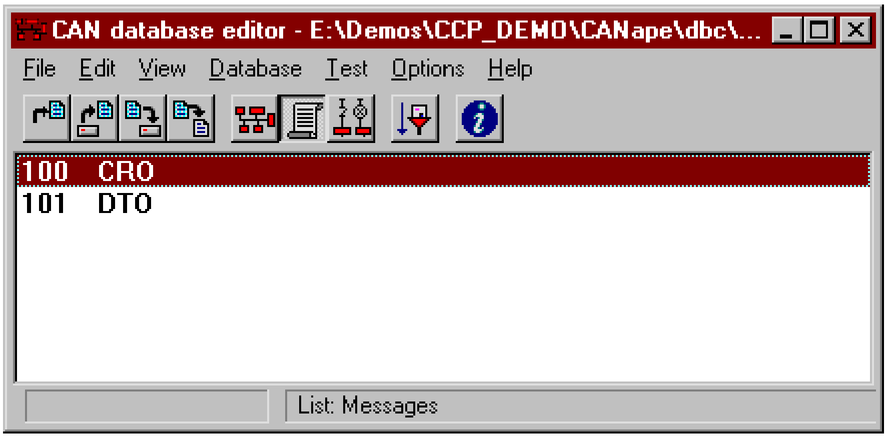
In section 3.1.1 we will show in detail how to extend the *.dbc file with two nodes and two messages.


CAN database editor - E:\Demos\CCP_DEMO\CANape\dbc\...
File Edit View Database Test Options Help

CCP_CANape
CCP_HC12

List: Network nodes

Figure 4: example list of nodes


CAN database editor - E:\Demos\CCP_DEMO\CANape\dbc\...
File Edit View Database Test Options Help

100 CRO
101 DTO

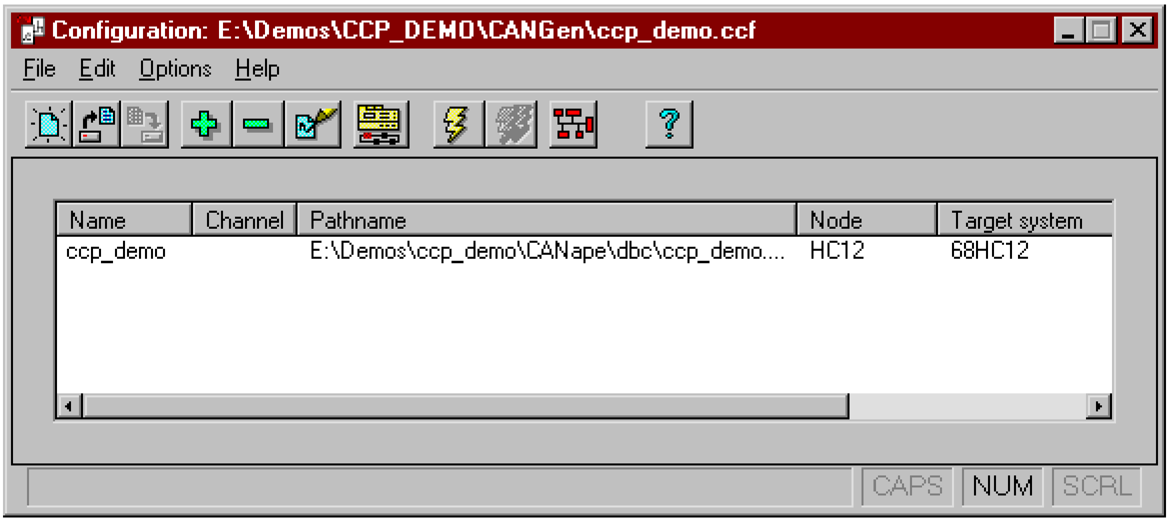
List: Messages

Figure 5: example list of messages

Note:
CRO’s and DTO’s have got a maximum size of 8 bytes in a CAN message (see *DLC Data Length Code* in the CAN specifications for further information).

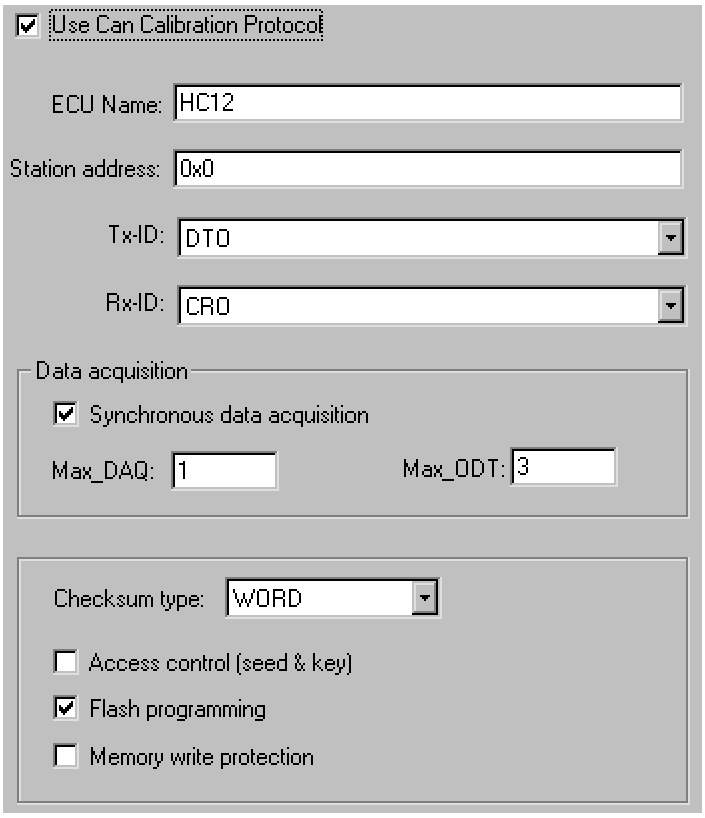
2.2.2 CANGen

CANGen, which comes up with the Vector CAN Driver delivery, is a tool which helps the application engineer to configure the Vector CAN and CCP Driver. In later sections we will see in detail how to use the CANGen tool to configure the Vector CAN and CCP Driver.



Name	Channel	Pathname	Node	Target system
ccp_demo		E:\Demos\ccp_demo\CANape\dbc\ccp_demo....	HC12	68HC12

Figure 6: example startup screen of the CANGen tool



☑ Use Can Calibration Protocol

ECU Name: HC12

Station address: 0x0

Tx-ID: DTO

Rx-ID: CRO

Data acquisition

☑ Synchronous data acquisition

Max_DAQ: 1

Max_ODT: 3

Checksum type: WORD

☐ Access control (seed & key)

☑ Flash programming

☐ Memory write protection

Figure 7: CANGen | CCP options

2.3 VECTOR CCP Driver

The Vector CCP Driver is delivered with two files:

- ccp.c : Vector CCP Driver Source Code
- ccp.h : Header file for the Vector CCP Driver

The Vector CCP Driver is configured with the CANGen tool, which generates the ***ccppar.h*** file. ccppar.h contains important definitions like which CRO (Command Receive Object) ID or DTO (Data Transmission Object) ID will be used, the number of DAQ lists, their size etc.

The interface between the CCP Driver and the CAN Driver has to be implemented as shown in Figure 8. The application calls the CCP Driver function *ccpInit()* once, e.g. after the initialization of the CAN Driver.

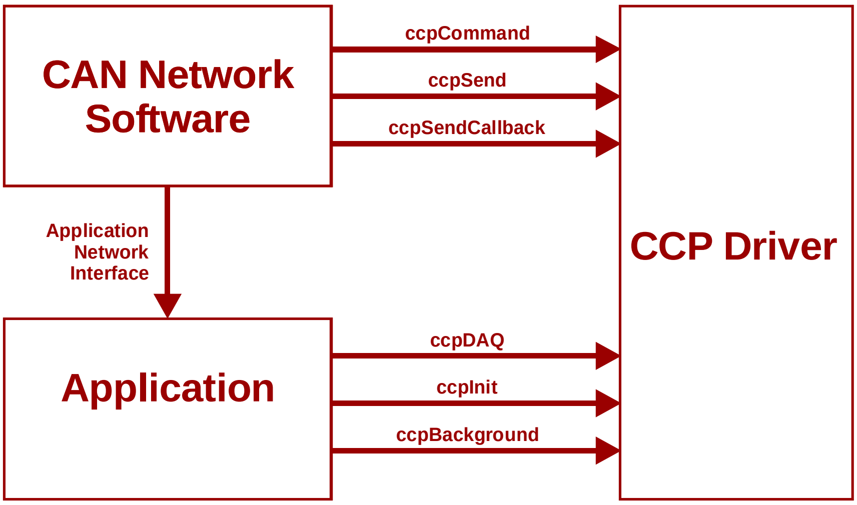


Figure 8: Functional Interface Overview Diagram

The CCP Driver uses the *ccpSend()* function to send DTO messages to the measurement and calibration system, e.g. CANape. After transmitting a DTO message, the CAN Driver has to call the CCP Driver function *ccpSendCallback()*. The CCP driver will not call *ccpSend()* again until *ccpSendCallBack()* has indicated the successful transmission of the previous message.

Some functions are not included in the *ccp.c* file and so have to be provided by the ECU developer. These functions are described in Section 2.3, *Description of Interface Functions* of the *CCP.PDF* documentation provided with the Vector CCP Driver. These interface functions should be written in a separate file, e.g. *ccp_can_interface.c* .

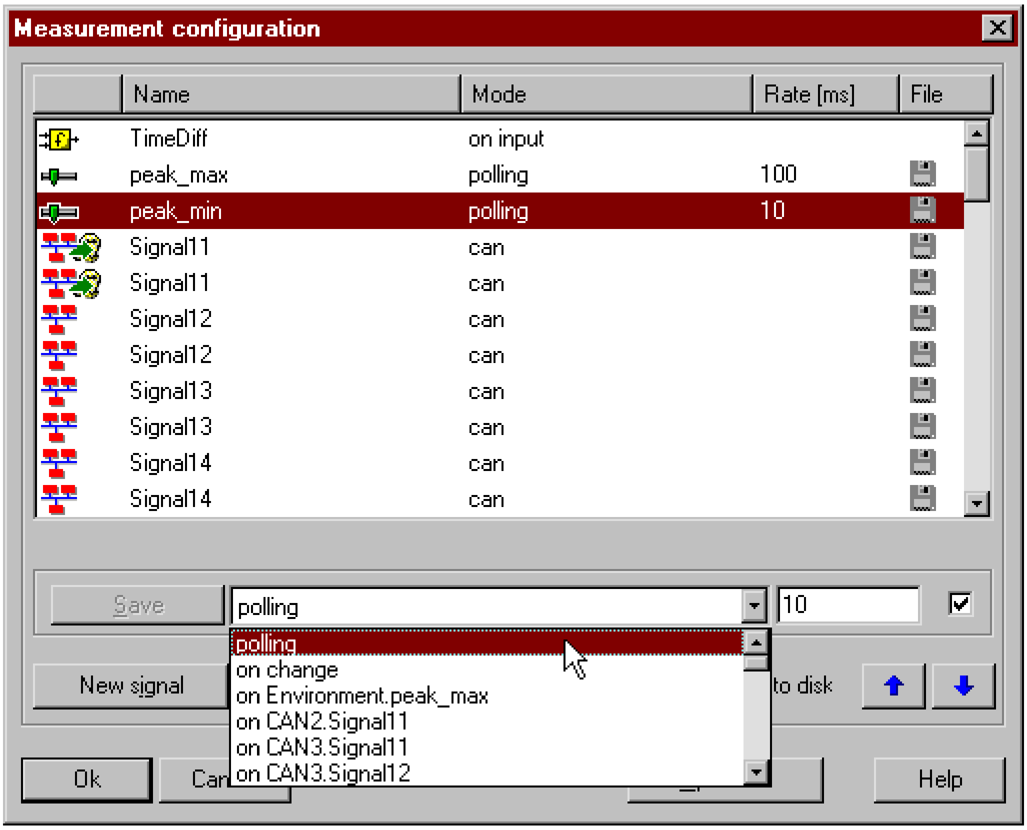
The first step will be to integrate the basic CCP Driver functionality in the ECU (see Chapter 3). If the communication between CANape and the ECU is set up properly, the other CCP options, like the synchronous data acquisition mode and the checksum calculation function, can be enabled in further steps (see Section 3.5)

2.3.1 Polling Mode

The polling mode is supported by the basic CCP command set. This means that if the CCP Driver is integrated in the ECU, the polling mode can be used immediately.

Note: No changes are required in the ECU software, but be sure the function *ccpInit()* is called one time when the main application is executed the first time.

The polling mode has to be selected in the CANape measurement configuration list. The user has to assign a data acquisition rate to each measurement signal (see Figure 9).



Measurement configuration

	Name	Mode	Rate [ms]	File
	TimeDiff	on input		
	peak_max	polling	100	💾
	peak_min	polling	10	💾
	Signal11	can		💾
	Signal11	can		💾
	Signal12	can		💾
	Signal12	can		💾
	Signal13	can		💾
	Signal13	can		💾
	Signal14	can		💾
	Signal14	can		💾

polling 10 ☑

polling
on change
on Environment.peak_max
on CAN2.Signal11
on CAN3.Signal11
on CAN3.Signal12

Figure 9: Configuration Of The Measurement List

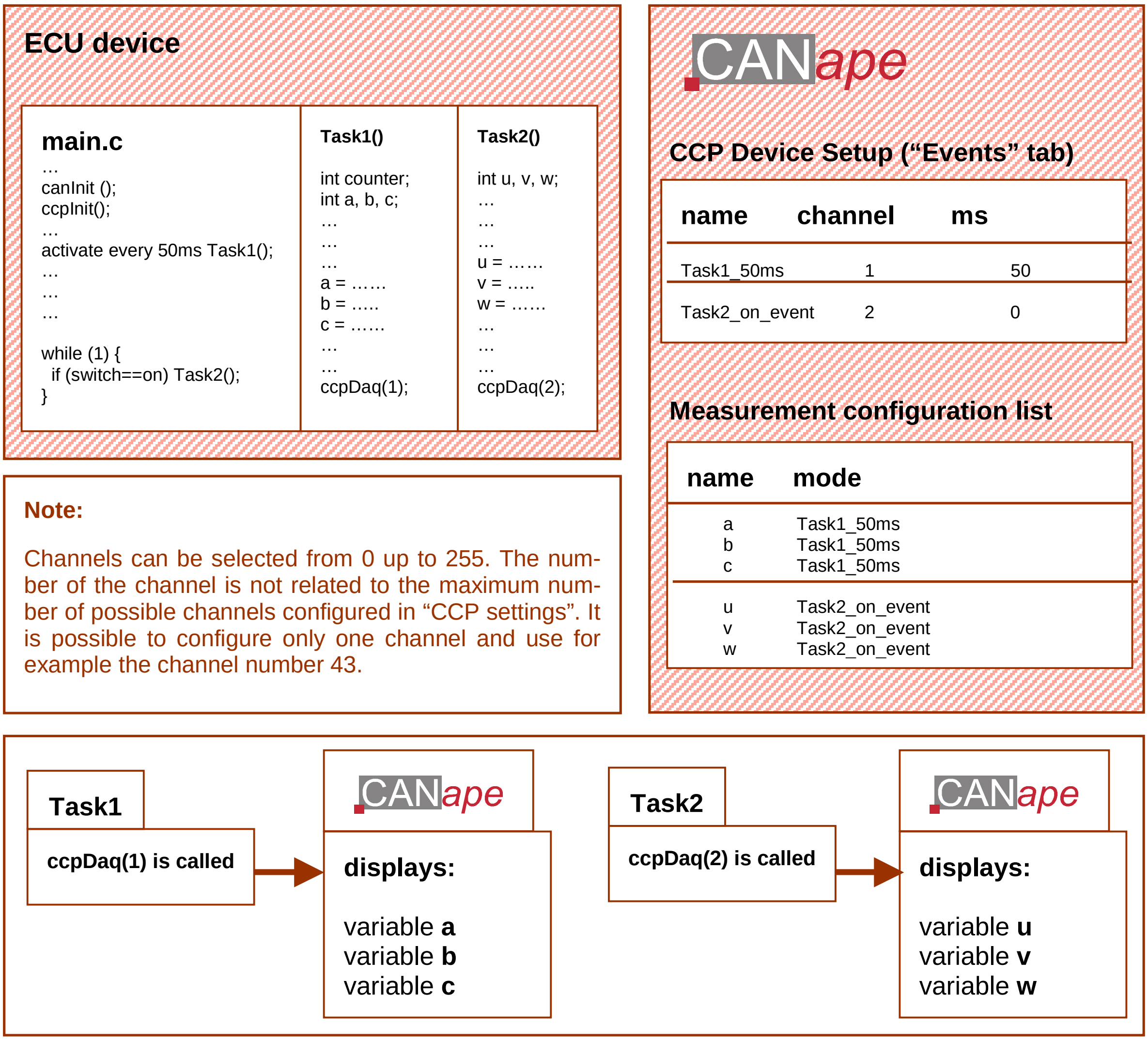
CANape automatically sends a request to the ECU according to the selected measurement rates and receives the measurement signal value as the response from the ECU, e.g. a current counter value etc. In Section 3.4 we will show how to enable the polling data acquisition mode in our HC12 demo sample.

2.3.2 Synchronous Data Acquisition Mode (DAQ)

The synchronous data acquisition mode has to be enabled in the ***ccppar.h*** file which is generated by the CANGen tool. Detailed steps are described in section 3.5. *Optional Features of the CCP Driver* in the *CCP.PDF* documentation provided with the Vector CCP Driver.

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 (see the example below) . This means that the measurement signals which have been assigned previously in the CANape measurement list are sent automatically to CANape during the ccpDaq(x) call. The number “x” represent the channel number used and the value of “x” can be a number between 0x00 and 0xFF.

The user has to define the desired “x” channel number for each *ccpDaq(x)* call in the program in the “CCP Device Setup” dialog in CANape. Finally, the user has to add the signals to measure from the different tasks in the measurement list and assign them to the correct task. In the example below, the signals *a,b and c* have to be assigned to *Task1_50ms* because these signals are used in Task 1. These signals will be transmitted automatically from the ECU to CANape every 50ms, during the *ccpDaq(1)* call. The signals *u, v and w* are assigned to *Task2_in_event*. Each time the specified event occurs, Task 2 is executed and only then will the signals u,v and w be transmitted automatically by the ECU to CANape.



ECU device

main.c

```
…
canInit ();
ccpInit();
…
activate every 50ms Task1();
…
…
…

while (1) {
  if (switch==on) Task2();
}
```

Task1()

```
int counter;
int a, b, c;
…
…
…
a = ……
b = …..
c = ……
…
…
ccpDaq(1);
```

Task2()

```
int u, v, w;
…
…
…
u = ……
v = …..
w = ……
…
…
…
ccpDaq(2);
```

Note:

Channels can be selected from 0 up to 255. The number of the channel is not related to the maximum number of possible channels configured in “CCP settings”. It is possible to configure only one channel and use for example the channel number 43.

CCP Device Setup (“Events” tab)

name	channel	ms
Task1_50ms	1	50
Task2_on_event	2	0

Measurement configuration list

name	mode
a	Task1_50ms
b	Task1_50ms
c	Task1_50ms
u	Task2_on_event
v	Task2_on_event
w	Task2_on_event

Task1

ccpDaq(1) is called

displays:

variable **a**
variable **b**
variable **c**

Task2

ccpDaq(2) is called

displays:

variable **u**
variable **v**
variable **w**

Figure 10: Example of an ECU program with two tasks:

In Section 3.5 we will see in detail how to enable the synchronous data acquisition mode in our HC12 demo sample.

2.3.3 Checksum Calculation

The use of a cache in CANape optimizes read accesses to the calibration memory of the ECU and is especially recommended if maps of dynamic size are used. Another benefit of the use of a cache in CANape is that the calibration engineer can also change parameters if the ECU is offline.

When the calibration engineer changes over from offline to online mode, a checksum is used to determinate whether the cache in CANape matches the ECU calibration RAM. If it does not match, the user is asked whether an upload or a download should be performed. In an upload, the contents of the calibration RAM are loaded from the ECU into cache. In a download the contents of the cache are copied into the ECU calibration RAM.

In order to use the cache in CANape, the ECU needs to be able to calculate a checksum on the calibration memory area. The checksum calculation can also be enabled in the ***ccppar.h*** file which is generated by the CANGen tool. Detailed steps are described in section 3.7.*Optional Features of the CCP Driver* in the *CCP.PDF* documentation provided with the Vector CCP Driver.

In Section 3.7 we will see in detail how to enable the checksum calculation in our HC12 demo sample and how to enable the cache in CANape.

2.3.4 Flash Kernel Download

The main task of an ECU is to perform calculations with sensor data or other signals in the RAM, while the main application is stored in the ROM or flash memory of the ECU. In running operation the user is able to change the behavior of the ECU via changing some parameters with help of an measurement and calibration tool, like CANape. The general disadvantage is that only RAM data can be changed, data stored in the flash memory can only be programmed with special flash routines.

To solve this problem it is possible to integrate flash routines into the code of the main ECU application. The disadvantage of this solution is that flash memory is wasted unnecessary, because these flash routines are not used very often. Another approach is the usage of a flash kernel. The flash kernel is loaded here by CANape into the microcontroller's RAM via CCP whenever the flash memory must be reprogrammed in the controller. The flash kernel contains all needed flash routines, its own CAN and CCP driver to communicate via the CAN interface with CANape.

In chapter 3.10 we will show how to enable the "Flash Kernel Download" in our HC12 CCP_DEMO example.

2.3.5 Seed & Key

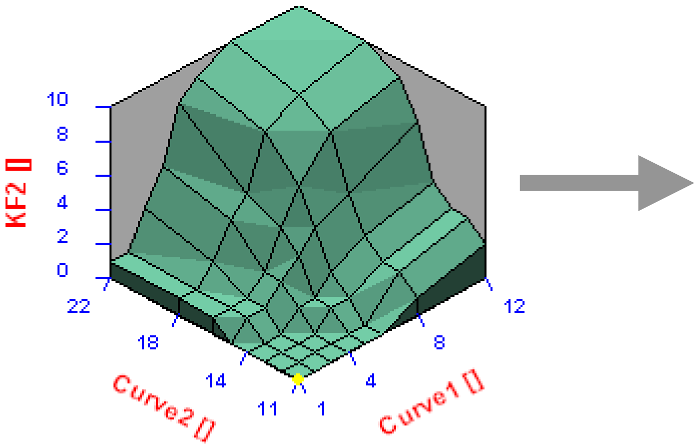
The application engineers are able to protect their ECU from editing parameters or variables by unauthorized persons. To enable this protection the "Seed & Key" option has to be enabled inside the CANGen tool.

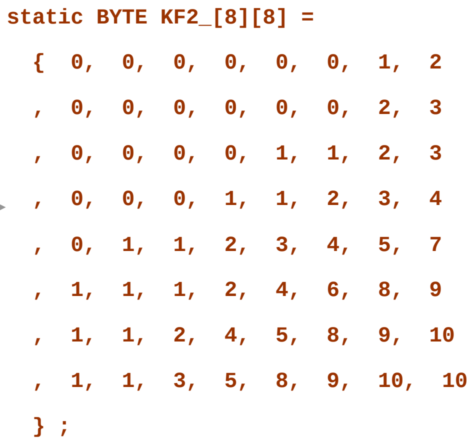
A seed is sent by the ECU to CANape when switching from Offline to Online Mode. CANape has to calculate the correct key with an special *.dll (see section 3.12) and return it to the ECU. Finally the ECU grants access if the key is correct.

In section 3.12 we will see in detail how to enable "Seed & Key" in our HC12 demo sample.

2.3.6 C Code Generation of Parameters

The C code generation is a new feature starting with CANape version 3.1.50 and enables the generation of C code for parameters, curves or maps etc. This feature is very useful when the parameter values were changed during a measurement and calibration session with CANape, and the user whishes consistent data in his C source code..





```
static BYTE KF2_[8][8] =
  { 0,  0,  0,  0,  0,  0,  1,  2
  , 0,  0,  0,  0,  0,  0,  2,  3
  , 0,  0,  0,  0,  1,  1,  2,  3
  , 0,  0,  0,  1,  1,  2,  3,  4
  , 0,  1,  1,  2,  3,  4,  5,  7
  , 1,  1,  1,  2,  4,  6,  8,  9
  , 1,  1,  2,  4,  5,  8,  9,  10
  , 1,  1,  3,  5,  8,  9,  10,  10
  } ;
```

Figure 11: result of a C code generation with a map

In Section 3.15 we will see in detail how to generate a C code file with our HC12 demo sample.

3 CCP_Demo Sample

In this chapter we will demonstrate how to interface the Vector CCP Driver to the Vector CAN Driver using the CCP Demo sample. The CCP Driver will be integrated into a Motorola HC12 microcontroller and we will enable step by step the different CCP options, like the synchronous data acquisition mode and the checksum calculation. At the same time we will configure CANape to make measurements in the polling mode, in the synchronous data acquisition mode, and we will enable the cache.

All the files you will need (e.g. the source code files, CANape project for the CCP Demo sample) are provided with the CCP Test demo sample.

The CCP_DEMO sample simulates an ECU program. The demo sample contains some parameters, curves and maps which can be changed by CANape. Some randomized variables, and calculated variables which depend on parameters, are also included in the demo sample. As you can see in Figure 12, Task 1 is called cyclically all 10 ms from the ECU. Every time Task 1 is run some calculations are performed. The input and output signals are simulated by variables inside the ECU, e.g. ampl, period, channel1, triangle. If the value of an input signal is changed via CANape, the effect can immediately be visualized in CANape.

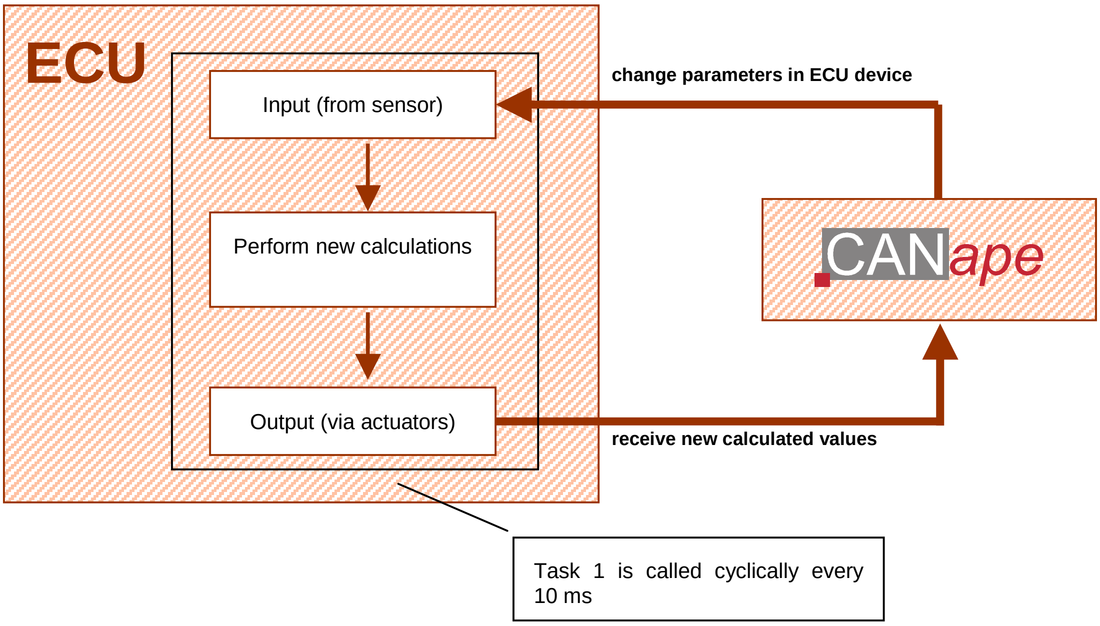
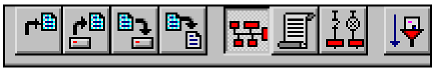

ECU
Input (from sensor)
Perform new calculations
Output (via actuators)
change parameters in ECU device
CANape
receive new calculated values
Task 1 is called cyclically every 10 ms

Figure 12: Task 1 of the ECU Demo Sample

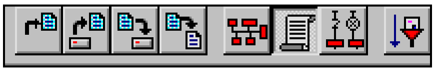
3.1 Configuration of the Vector CCP Driver

3.1.1 Adding CRO and DTO message in an existing *.dbc file

- ✓ start CANdb Editor
- ✓ open your database file (e.g. ccp_demo.dbc)
- ✓ select “Display network nodes of the database”



- ✓ right-click in the empty space and click “New network node…”
- ✓ enter name (e.g. CCP_CANape), click “Ok”
- ✓ right-click in the empty space and click “New network node…”
- ✓ enter name (e.g. CCP_HC12), click “Ok”
- ✓ select “Display messages of database”

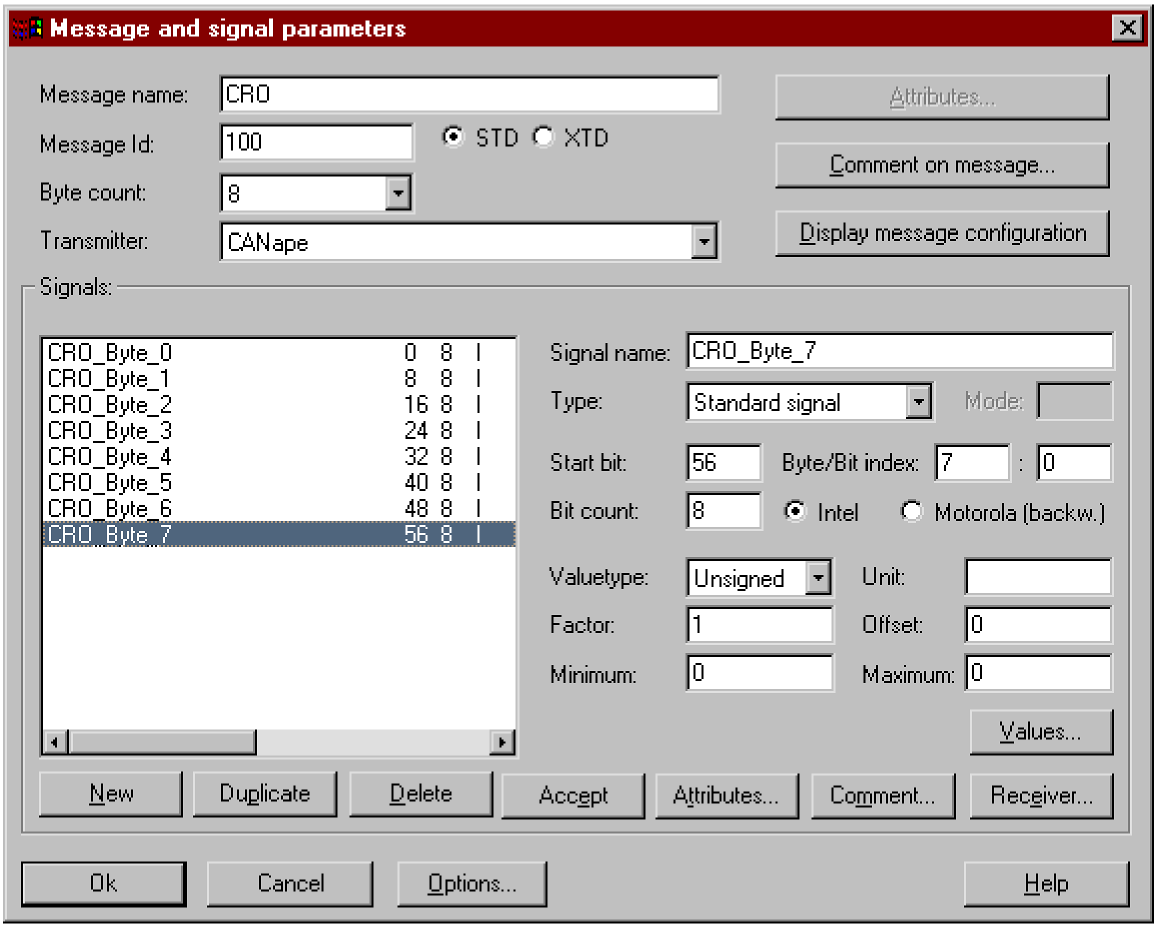


- ✓ right-click in the empty space and click “New message…”
- ✓ enter a message name (e.g. CRO)
- ✓ enter a message id (e.g. 100)
- ✓ select byte count 8
- ✓ CRO is transmitted by node “CCP_CANape”
- ✓ now enter a signal name (e.g. CRO_Byte_x),
  click “Receiver” and select “CCP_HC12”

  do this 8 times and count up the start bit, “Byte count” and “Bit count” are always set to 8

- ✓ be sure that each signal has got its right receiver!

Note:
node “CCP_CANape” sends Command Receive Object (CRO) messages,
node “CCP_HC12” transmits the wished parameters by Data Transmission Object (DTO).



Message name: CRO
Message Id: 100 ● STD ○ XTD
Byte count: 8
Transmitter: CANape

Signals:

CRO_Byte_0	0	8	I
CRO_Byte_1	8	8	I
CRO_Byte_2	16	8	I
CRO_Byte_3	24	8	I
CRO_Byte_4	32	8	I
CRO_Byte_5	40	8	I
CRO_Byte_6	48	8	I
CRO_Byte_7	56	8	I

Signal name: CRO_Byte_7
Type: Standard signal Mode:
Start bit: 56 Byte/Bit index: 7 : 0
Bit count: 8 ● Intel ○ Motorola (backw.)
Valuetype: Unsigned Unit:
Factor: 1 Offset: 0
Minimum: 0 Maximum: 0

Figure 13: CAN message “CRO” and its signals

- ✓ follow the instructions above to create the message DTO in node “HC12”, but change transmitter to “HC12”, receiver to “CANape”, ID=101 and enter also eight data bytes as seen before.
- ✓ be sure that each signal has got its right receiver!
- ✓ save the current database (e.g. ccp_demo.dbc)

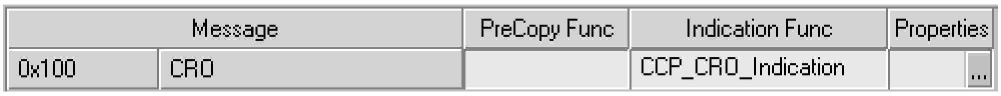
3.1.2 CANGen SETUP – General Settings

- ✓ start **CANGen**
- ✓ click “Add channel” in the toolbar
- ✓ select your data base, in our example "ccp_demo.dbc"
- ✓ select the node "HC12", click "Ok"
- ✓ click “Generation Options” in the toolbar
- ✓ tab "Overview" shows the messages which are set in the data base file "ccp_demo.dbc" change the directories if necessary
- ✓ leave tab "CAN driver" unchanged
- ✓ enter tab "CAN driver (advanced)" and change
  Register Block Address to 0x000, because our register is not remapped and begins at $000 hex

The following configuration is done automatically by the CANGen tool, there is no need to do this steps manually.

Attention: The Indication/Confirmation functions are NOT displayed!

- ✓ select tab "Receive messages | Functions" and enter: CCP_CRO_Indication



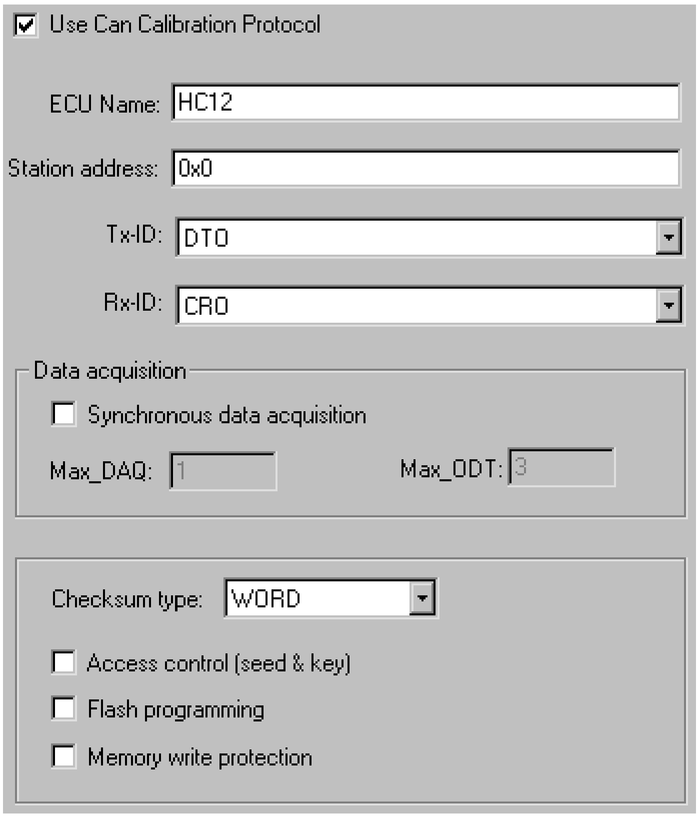
Message		PreCopy Func	Indication Func	Properties
0x100	CRO		CCP_CRO_Indication	...

Note:
CCP_CRO_Indication is an implemented function in **can_ccp.c** and it is used for ccp-can-interfacing purpose.

- ✓ select tab "Send messages / Functions" and enter: CCP_DTO_Confirmation

Note:
CCP_DTO_Confirmation is an implemented function in **can_ccp.c** and it is used for ccp-can-interfacing purpose.

- ✓ now change to tab “CCP options” and check “Use Can Calibration Protocol”



☑ Use Can Calibration Protocol

ECU Name: HC12

Station address: 0x0

Tx-ID: DTO

Rx-ID: CRO

Data acquisition

☐ Synchronous data acquisition

Max_DAQ: 1 Max_ODT: 3

Checksum type: WORD

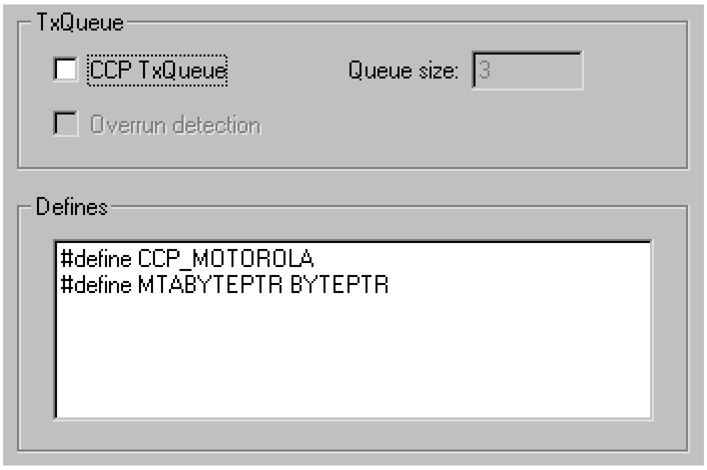
☐ Access control (seed & key)

☐ Flash programming

☐ Memory write protection

- ✓ enter the name of the ECU and set Transmit/Receive Ids
- ✓ add the following line in “Defines”:

#define CCP_MOTOROLA
#define MTABYTEPTR BYTEPTR



TxQueue

☐ CCP TxQueue Queue size: 3

☐ Overrun detection

Defines

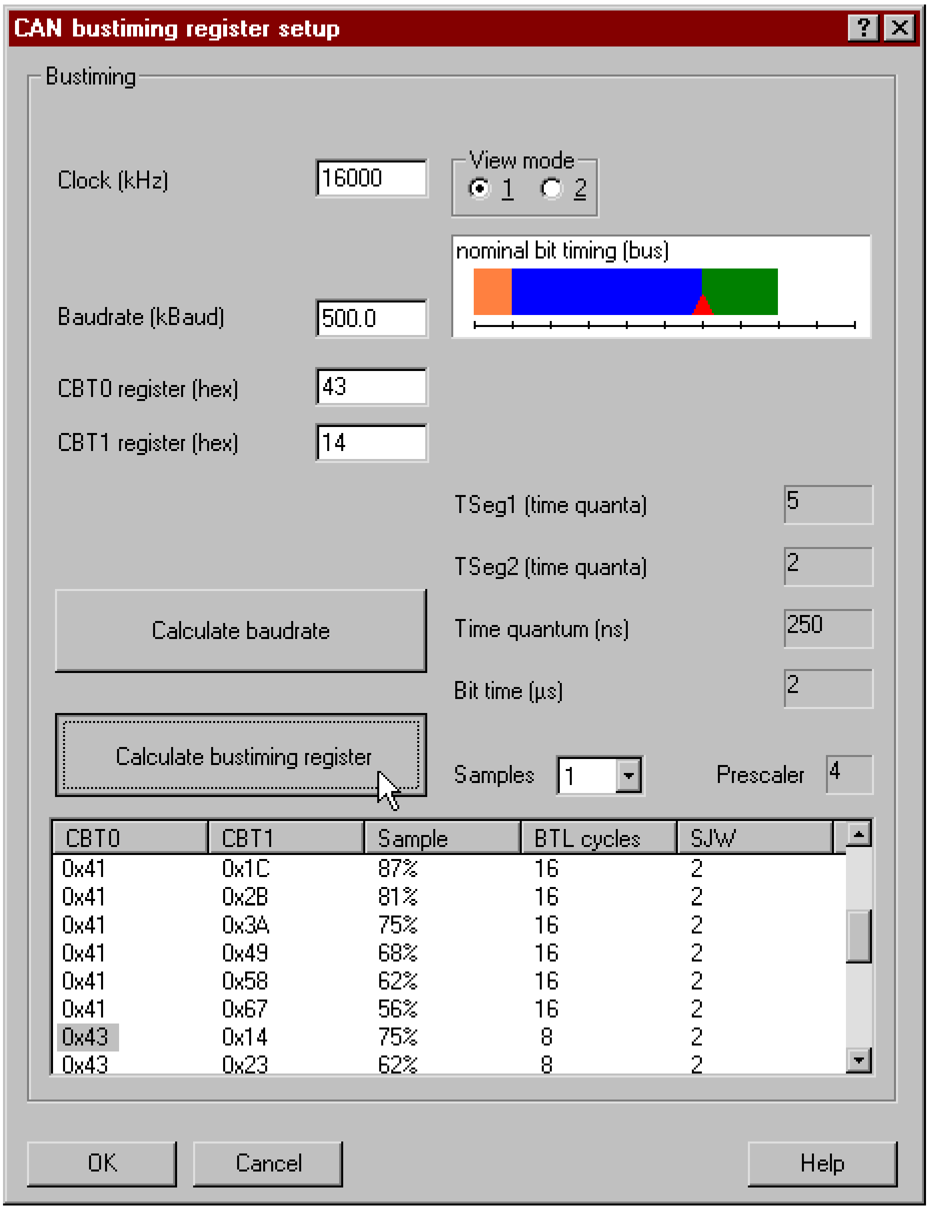
```
#define CCP_MOTOROLA
#define MTABYTEPTR BYTEPTR
```

Note:
BYTEPTR is defined as: unsigned char*
MTABYTEPTR is used in can_ccp.h

✓ change to tab “Init Registers”

✓ click “Bustiming registers…” , be sure the right baud rate is entered
(in our example we use 500 kBaud),

✓ click “Calculate bustiming registers”, click “Ok”



CAN bustiming register setup

Bustiming

Clock (kHz): 16000

View mode: ● 1 ○ 2

Baudrate (kBaud): 500.0

CBT0 register (hex): 43

CBT1 register (hex): 14

TSeg1 (time quanta): 5

TSeg2 (time quanta): 2

Time quantum (ns): 250

Bit time (µs): 2

Samples: 1

Prescaler: 4

CBT0	CBT1	Sample	BTL cycles	SJW
0x41	0x1C	87%	16	2
0x41	0x2B	81%	16	2
0x41	0x3A	75%	16	2
0x41	0x49	68%	16	2
0x41	0x58	62%	16	2
0x41	0x67	56%	16	2
0x43	0x14	75%	8	2
0x43	0x23	62%	8	2

✓ click "Apply" or "Ok"

✓ click "Generate"

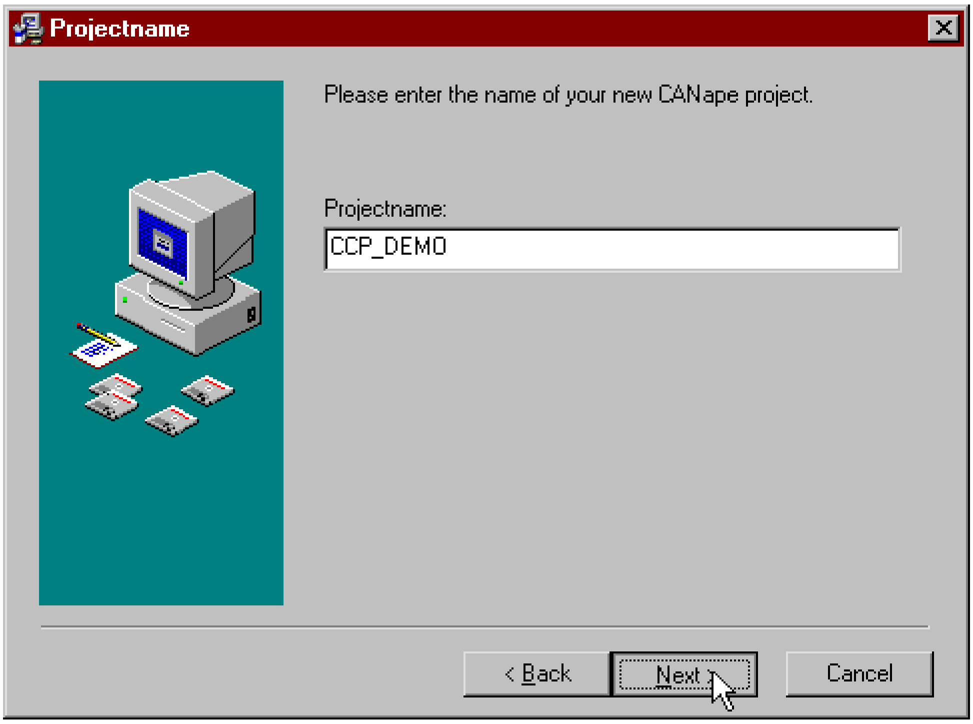
- ✓ now put all generated *.h files (project.h, ccppar.h, can_cfg.h, v_cfg.h) into your INCLUDE folder if not done automatically

- ✓ be sure the generated project.c (in our example hc12.c) file is in your SOURCE folder

3.2 Creation of a New CANape Project

In this chapter we will create a new CANape project for the CCP DEMO sample. We will start CANape and then add the HC12 with the CCP DEMO sample in the CANape device list.

- ✓ choose "Create new project" in the windows start menu in the CANape program folder
- ✓ click "Next" and enter a name for your project, e.g. CCP_DEMO

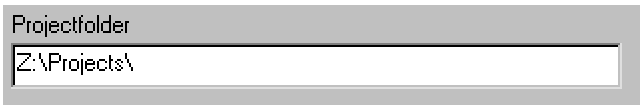


Projectname

Please enter the name of your new CANape project.

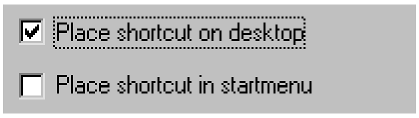
Projectname: CCP_DEMO

- ✓ in the next window select a path for your new project



Projectfolder: Z:\Projects\

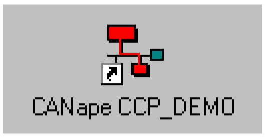
- ✓ be sure the checkbox "Place shortcut on Desktop" is checked; uncheck "Startmenu" if necessary



☑ Place shortcut on desktop

□ Place shortcut in startmenu

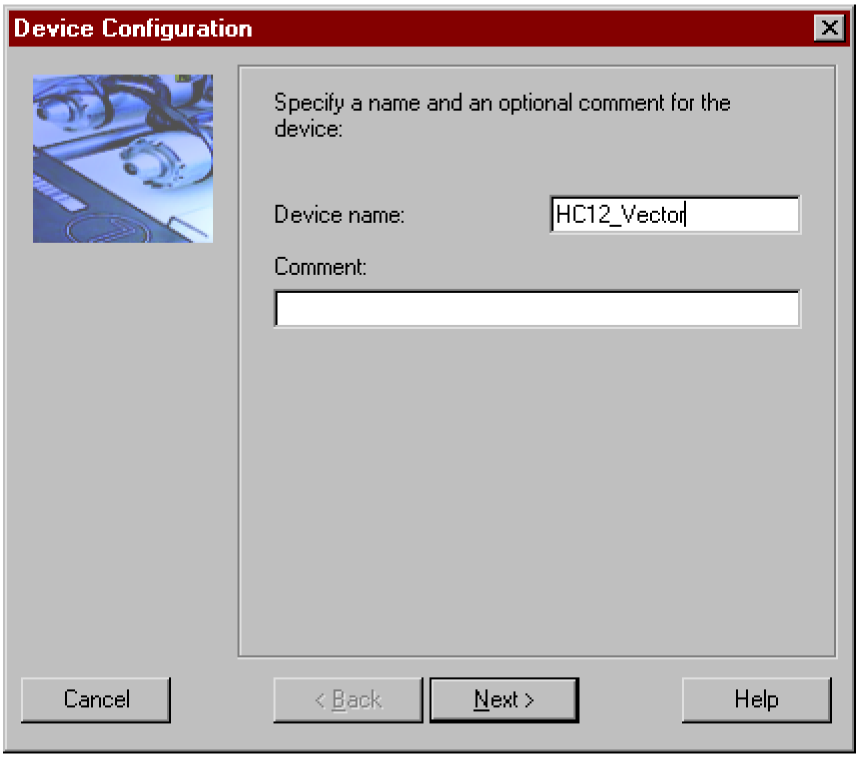
- ✓ click "Next". Do NOT “Start CANape immediately“, but instead click “Finish“
- ✓ on the Windows desktop a new icon with the name of your project has been created:


CANape CCP_DEMO

- ✓ double click this icon – this will start CANape

Now we are ready to add a new device (ECU):

- ✓ in CANape, press on the symbol in the toolbar or select the menu command “Options | Device configuration” to add a new device in the CANape device list
- ✓ press the "New" button
- ✓ enter a name for your device, e.g. HC12_Vector


Device Configuration

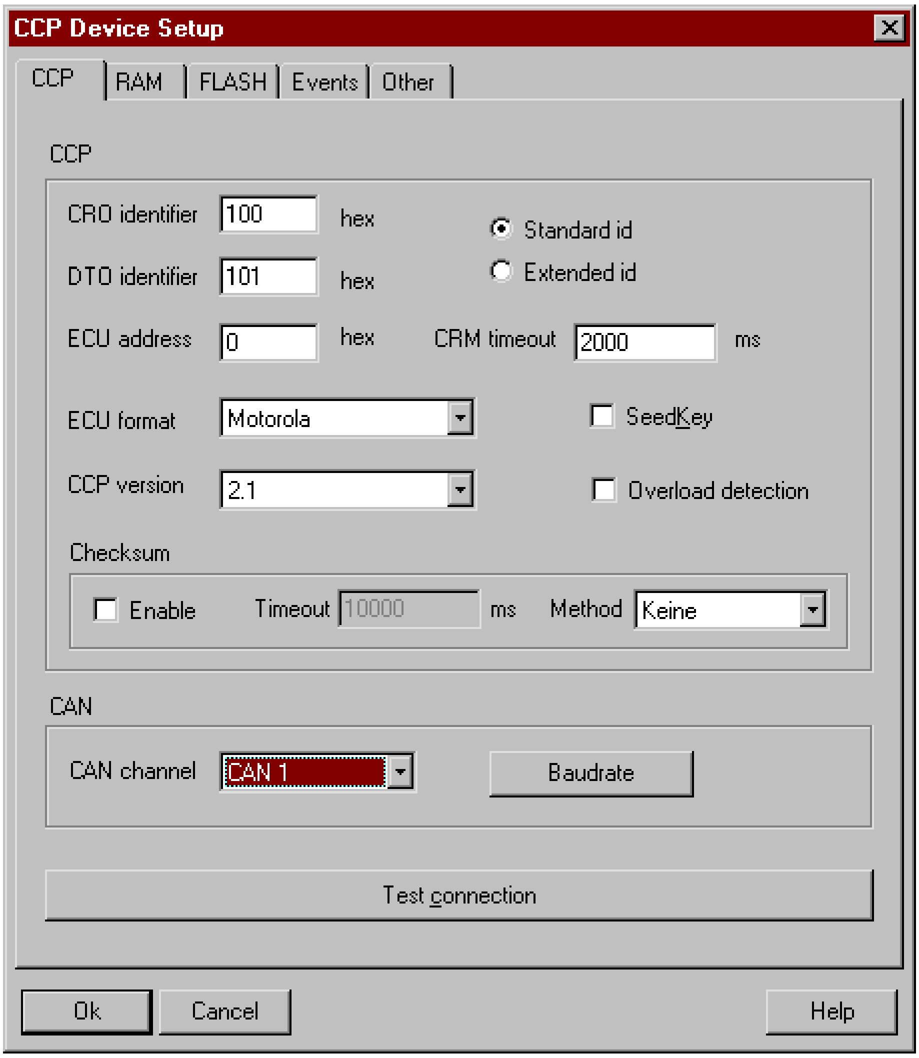
Specify a name and an optional comment for the device:

Device name: HC12_Vector

Comment:

- ✓ click “Next”

- ✓ now you have to select the type of the driver used in your device.
  Select CCP press the tab key to enter the drivers options and click the “Driver configuration” button
- ✓ CRO-Identifier: enter the ID for the CCP driver (in our example: 100h)
- ✓ DTO-Identifier: enter the ID for the CCP driver (in our example: 101h)
- ✓ ECU-Address: enter the ECU address (in our example: 0)
- ✓ ECU-Format: Motorola
- ✓ uncheck "Seed+Key", "Overload detection" and Checksum “Enable”. The tab page now should look like this:



CCP Device Setup

CCP | RAM | FLASH | Events | Other

CCP

CRO identifier: 100 hex

DTO identifier: 101 hex

● Standard id
○ Extended id

ECU address: 0 hex

CRM timeout: 2000 ms

ECU format: Motorola

CCP version: 2.1

□ SeedKey

□ Overload detection

Checksum

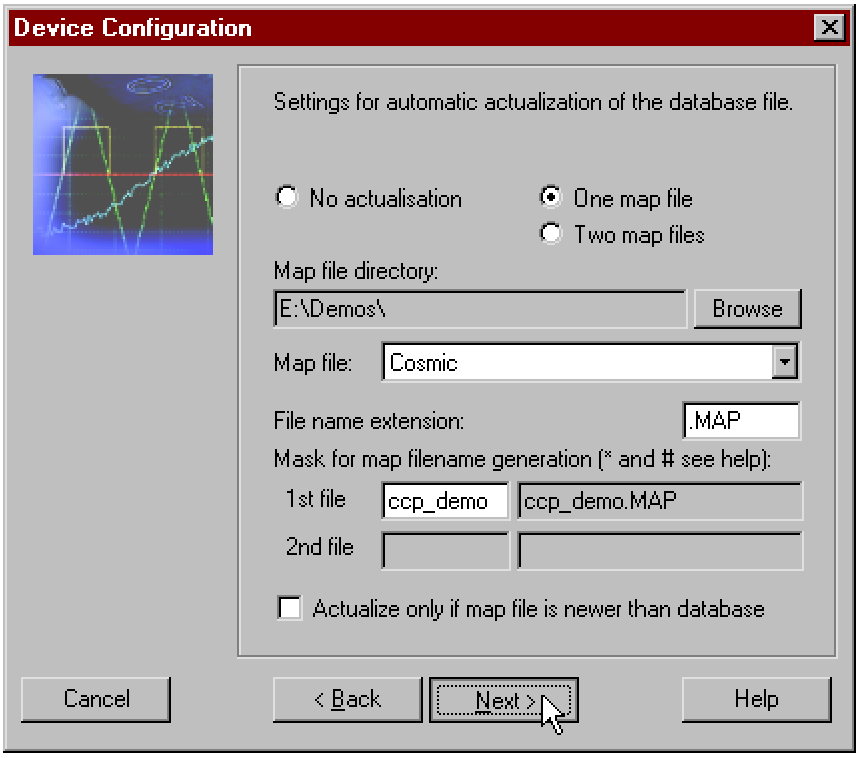
□ Enable   Timeout: 10000 ms   Method: Keine

CAN

CAN channel: CAN 1

- ✓ select the CAN channel and configure the baud rate (e.g. CAN1, 500kBit/s)
- ✓ close the dialog with “Ok” and click “Next”
- ✓ use the default settings on the next dialog page and click “Next”
- ✓ select "one map file" and select the map file directory with the “Browse” button

- ✓ map file: select COSMIC format (our CCP Test demo sample was compiled with a COSMIC C compiler)
- ✓ 1^{st} file: enter the name of the map file "ccp_test". The extension is automatically set (the default extension name is .MAP)



Device Configuration

Settings for automatic actualization of the database file.

○ No actualisation ● One map file ○ Two map files

Map file directory: E:\Demos\

Map file: Cosmic

File name extension: .MAP

Mask for map filename generation (* and # see help):

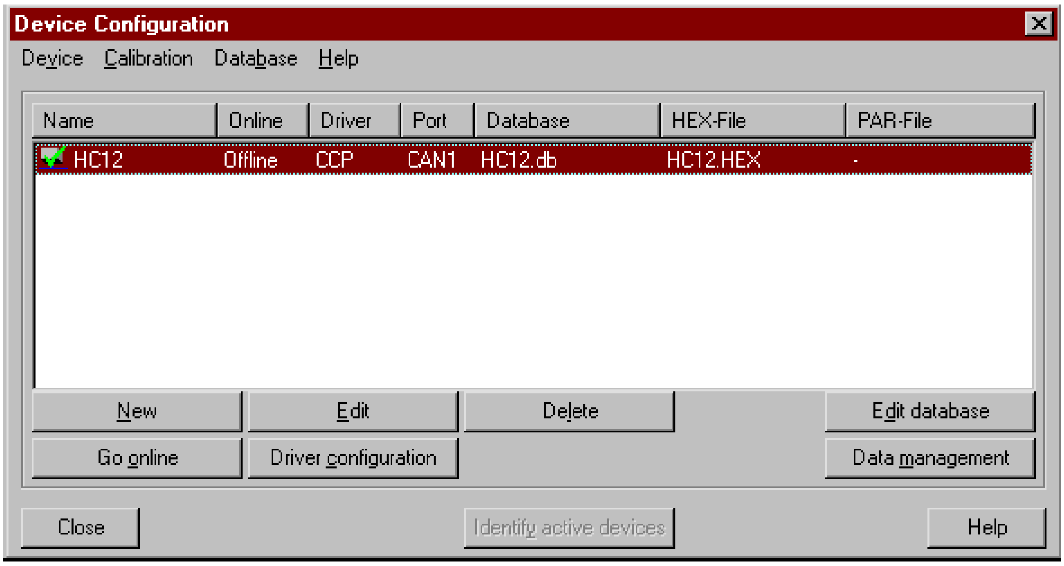
1st file: ccp_demo ccp_demo.MAP

2nd file:

□ Actualize only if map file is newer than database

Note:
You need a linker map file to create a controller description file (*.db or *.a2l). The linker map file contains all the measurement and calibration objects with their addresses and, depending on the map format, also the data type from the ECU.

- ✓ click “Next”
- ✓ The device configuration dialog for Version 3.2 now displays a window asking you to specify the directory for the device’s hex file. Accept the default settings. Click “next”
- ✓ The Project Wizard displays a summary of your settings. Click “Back” to go back and change them if you wish, or click “Ok”
- ✓ CANape tries now to get a connection to the ECU and opens a dialog to ask if a database should be imported. Click “No”
- ✓ in the device list the new created device is displayed with a green check symbol



Name	Online	Driver	Port	Database	HEX-File	PAR-File
✓ HC12	Offline	CCP	CAN1	HC12.db	HC12.HEX	-

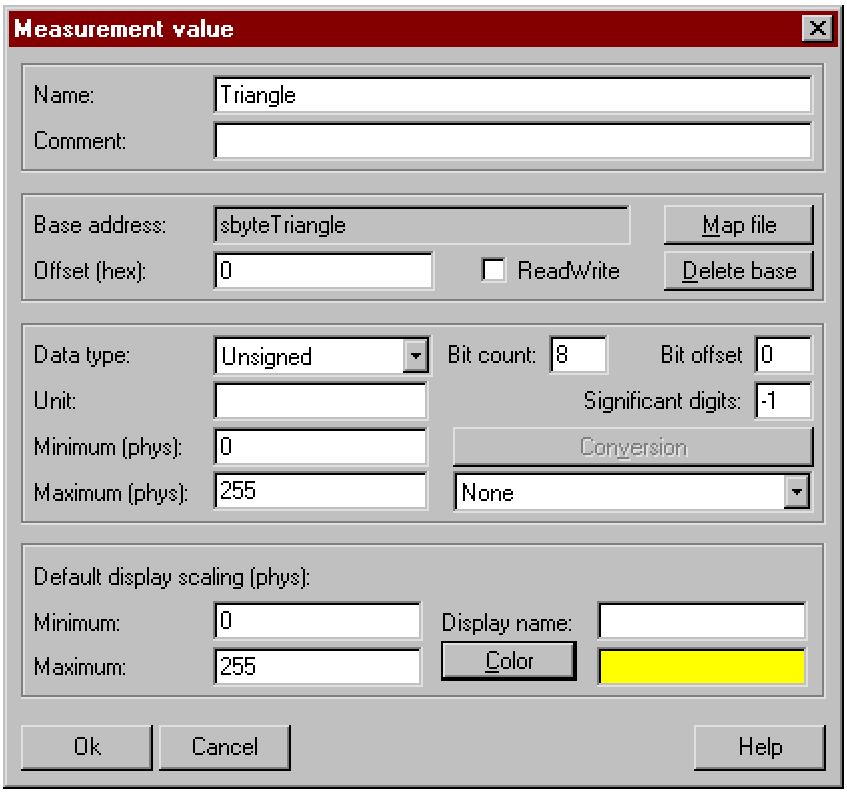
- ✓ click "Close"

3.3 Adding a Measurement Signal to the Controller Database

CANape has created an empty controller database. In this section we will add a measurement signal into the controller database. A measurement signal can be an output variable from the ECU device. In our example we will add the “Triangle” signal. Other signals and parameters can be added into the ECU controller database in a similar manner. For more details, see the corresponding chapter in the CANape User Manual.

✓ Click on the DB symbol in the toolbar to open the controller database with the database editor

✓ right-click in the right empty space and select “New | Parameter or measurement value” If you are using Version 3.2, select “New -> Measurement value”



Measurement value

Name: Triangle

Comment:

Base address: sbyteTriangle

Offset (hex): 0

□ ReadWrite

Data type: Unsigned

Bit count: 8

Bit offset 0

Unit:

Significant digits: -1

Minimum (phys): 0

Maximum (phys): 255

None

Default display scaling (phys):

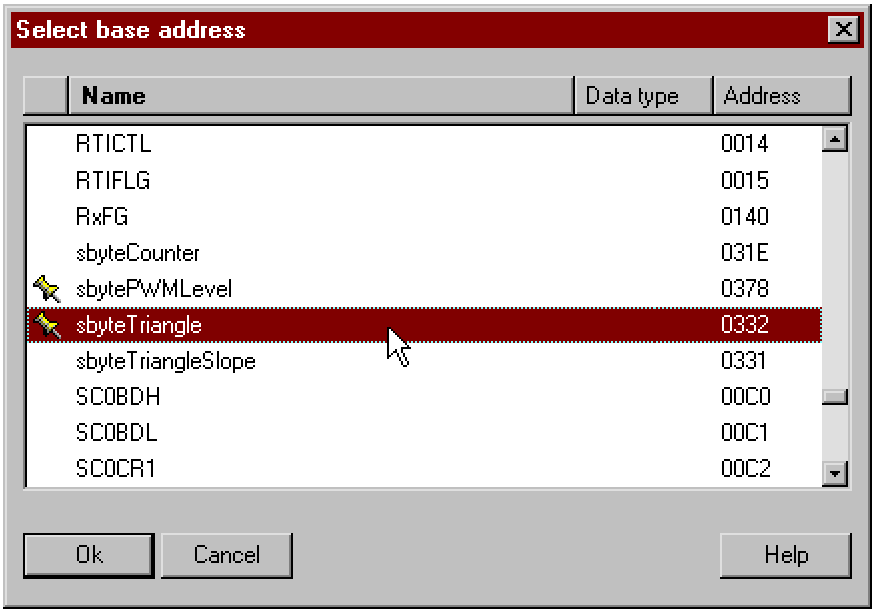
Minimum: 0

Maximum: 255

Display name:

✓ enter a name for the value (e.g. Triangle) and a comment if you wish

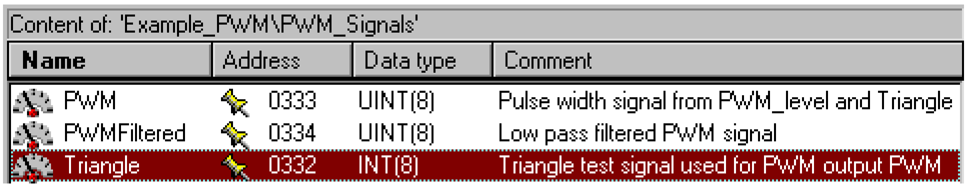
- ✓ click “Mapfile” to select a variable



Select base address

Name	Data type	Address
RTICTL		0014
RTIFLG		0015
RxFG		0140
sbyteCounter		031E
sbytePWMLevel		0378
sbyteTriangle		0332
sbyteTriangleSlope		0331
SC0BDH		00C0
SC0BDL		00C1
SC0CR1		00C2

- ✓ select the variable, here: _triangle
- ✓ click “Ok”. You should see something like this on the right side:



Content of: 'Example_PWM\PWM_Signals'

Name	Address	Data type	Comment
PWM	0333	UINT(8)	Pulse width signal from PWM_level and Triangle
PWMFiltered	0334	UINT(8)	Low pass filtered PWM signal
Triangle	0332	INT(8)	Triangle test signal used for PWM output PWM

- ✓ exit the database editor with .

3.4 Measuring Signals in Polling Mode

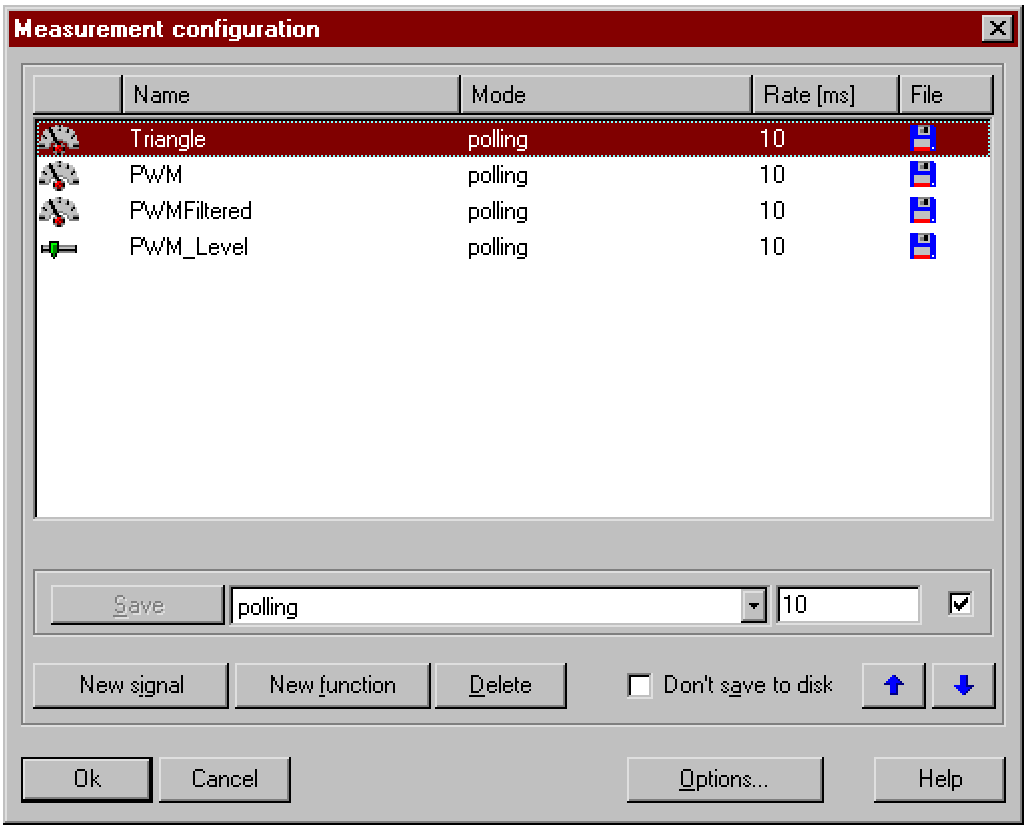
In this chapter we will add the “Triangle” signal in the CANape measurement list and acquire its data in the polling mode. Then we configure a graphic window to display the triangle signal during the running measurement.

- ✓ open the measurement list with the ▤ symbol in the toolbar
- ✓ click "New signal" and select in the controller database "triangle" .
  Press "Apply",

  ✓ Apply

  close the controller database window. The “Triangle” signal will now appear in the Measurement configuration list,

  be sure that the data acquisition mode is set to “polling”



Measurement configuration

	Name	Mode	Rate [ms]	File
	Triangle	polling	10	💾
	PWM	polling	10	💾
	PWMFiltered	polling	10	💾
	PWM_Level	polling	10	💾

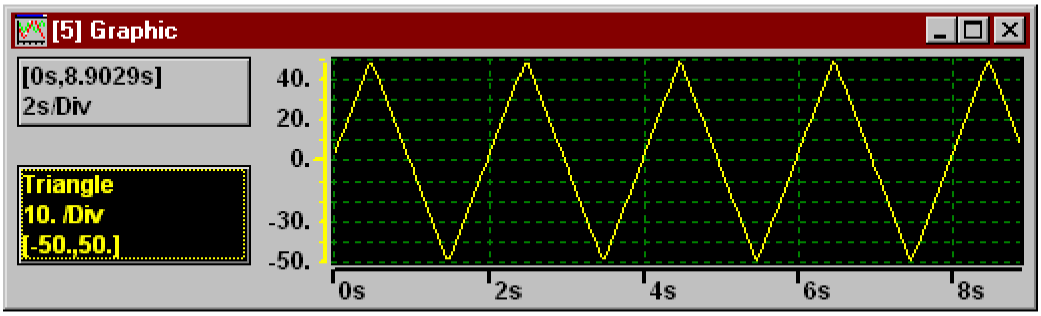
Save | polling | 10 | ☑

- ✓ click "Ok"
- ✓ create a new graphic window in CANape by right-clicking and selecting the entry “Graphic window”

- ✓ add the signal "Triangle" by right-clicking the empty graphic window and selecting the entry “Insert measurement signal”

- ✓ start the measurement by pressing the F9 key or the ⚡ symbol in the toolbar

- ✓ now a triangle signal should be displayed in the graphic window



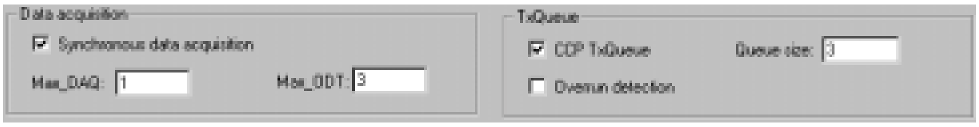
- ✓ The measurement can be stopped by pressing the ESC key or the ● symbol in the toolbar

3.5 Enabling the Synchronous Data Acquisition Mode (DAQ)

In this section we will enable the synchronous data acquisition mode in the CCP Demo sample. In this example we will enable two DAQ lists with three object descriptor tables (ODT) and we will use the send queue. With this configuration, we can measure a maximum of 42 bytes (2 * 3 * 7 bytes) at two different time rates.

The synchronous data acquisition mode has to be enabled in the *ccppar.h* file which is generated by the CANGen tool:

- ✓ open CANGen with the previous configuration (in our example ccp_demo.ccf)
- ✓ click “Generation Options” in the toolbar
- ✓ change to tab “CCP options” and enable the checkbox “Data acquisition”, enter two DAQ lists and 3 ODT’s



Data acquisition
☑ Synchronous data acquisition
Max_DAQ: 1
Max_ODT: 3

TxQueue
☑ CCP TxQueue
Queue size: 3
☐ Overrun detection

- ✓ enable the transmit queue and enter a size of 3
- ✓ apply and save all changes, generate the new files

We have to add the *ccpDaq(x)* function calls in our ECU source code at all the places where the ECU has to transmit the measurement signals. In our CCP DEMO sample we will call the *ccpDaq(x)* function only in one place, in the interrupt service routine for Timer_0. The timer interrupt event has been programmed in our example to occur every 10 ms, which means that the timer interrupt service routine is also executed every 10 ms.

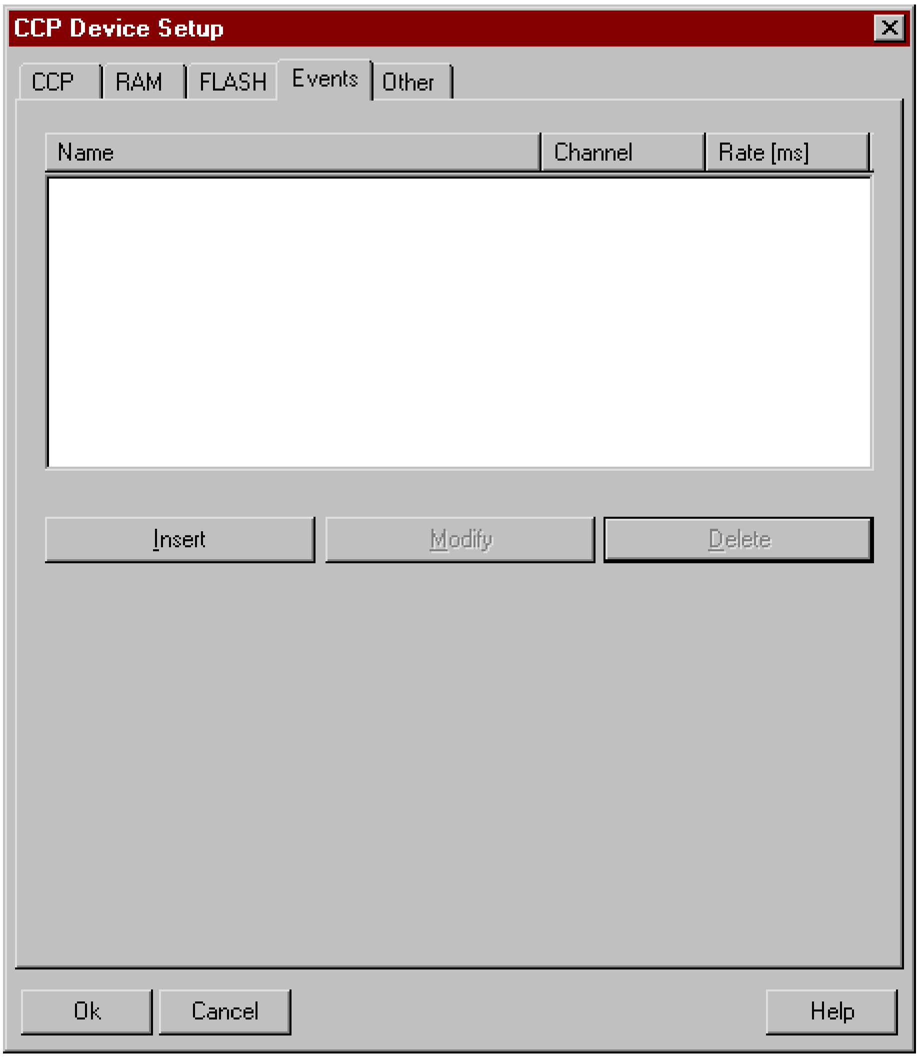
```
// -----------------------------------------------------------------------------
// Interrupt Service Routine for Timer_0 //
// -----------------------------------------------------------------------------
@interrupt void _timer_0(void)
{
 TC0 = TC0 + counter;           // incr. TimerCounter
 TFLG1 = 1;                     // With this micro, writing a one to the bit
                               // clears the interrupt flag so the interrupt can occur

/*  the TFLG1 flag must be cleared before ccpDaq() is
called, because  this subroutine enables all interrupts
at the end, at this point in the code the TFLG1 flag
is still 1, so the interrupt by the timer occurs again.
Normally the interrupts are enabled at the end of the
interrupt service routine of timer_0.
*/

 ecuCyclic();                                  // calculate new values
 ccpDaq(1);                                    // transmit new values
};
// -----------------------------------------------------------------------------
```

Now you have to recompile the project and re-program the HC12 with the new code.

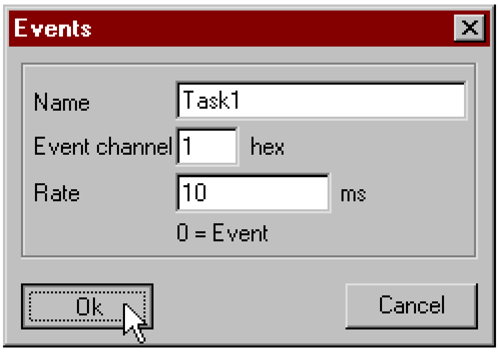
3.6 Using the Synchronous Data Acquisition Mode in CANape

We now have to define in CANape all the used synchronous data acquisition event channels in the ECU. Then we have to configure the measurement mode of the signals in the measurement list.

- ✓ start CANape with a double click on the CANape icon on your desktop
- ✓ open the CCP Device Setup dialog with "Options | Driver configurations…"
- ✓ select the tab "Events"



- ✓ click “Insert” and enter a name for a new synchronous data acquisition event channel (e.g. Task1) with the channel number 1 (in this example we only use one channel as described before)



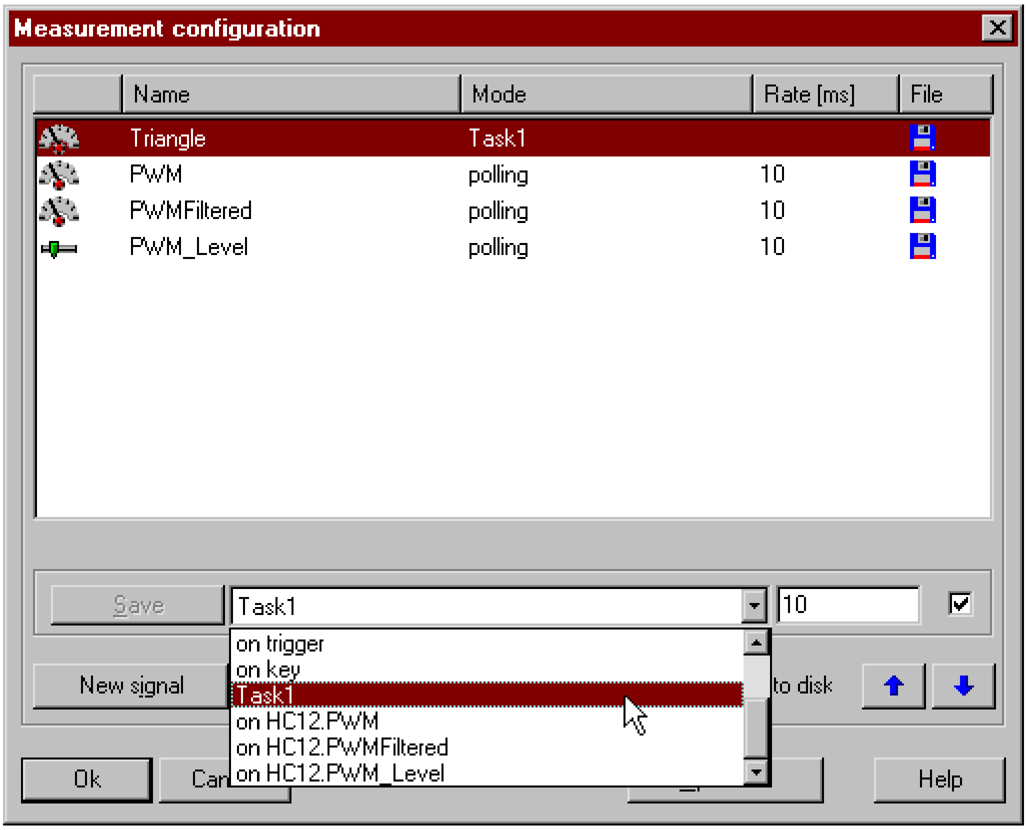
Name: Task1

Event channel: 1 hex

Rate: 10 ms

0 = Event

- ✓ set “Rate” to 10ms (ccpDaq(1) is executed every 10ms in the timer interrupt service routine)
- ✓ click “Ok”, click “Ok” again to close the CCP Device Setup dialog
- ✓ open the measurement list with the symbol in the toolbar
- ✓ change the measurement mode for the “Triangle” signal from “polling” to “Task1”. Each time the function **ccpDaq(1)** is called in the ECU application, the value of the variable “Triangle” is transmitted via CAN and the new value is displayed in CANape.



Name	Mode	Rate [ms]	File
Triangle	Task1		💾
PWM	polling	10	💾
PWMFiltered	polling	10	💾
PWM_Level	polling	10	💾

Task1 ▾ 10 ☑

on trigger
on key
Task1
on HC12.PWM
on HC12.PWMFiltered
on HC12.PWM_Level

Note: By inserting an event with a fixed rate (here: 10 ms) the new data acquisition mode "cyclic" is automatically created.

The user can assign one of these data acquisition modes for each signal in the measurement signal list:

polling: In this mode the user has to define the data transmission rate. CANape sends a query to the ECU at the specified rate.

Note: In the polling mode, the bus load is greater than in all the other modes and the values are not sampled synchronously.

Task1: In this mode the data transmission rate is determined by the ECU device itself. The rate is not set here. Instead, the new values are sent to CANape every time **ccpDaq(1)** is called in the ECU program. In our example the ECU will send the new signal values every 10 ms.

cyclic: In this mode the data transmission rate is determined by the ECU, but the user has to enter a rate as a multiple of the default transmission rate in milliseconds. CANape calculates a prescaler value and configures a new DAQ list with this value. Each time **ccpDaq(1)** is called, the prescaler value is decremented. If the prescaler value is 0, then **ccpDaq(1**) transmits the measurement signal values.

Note:
The ECU needs a separate DAQ list for each new data transmission rate used in-the measurement list.

.

Example for 'cyclic' mode:

When TASK1 is created 10ms is set as the default data transmission rate
Data transmission rate set by the user: 50 ms using 'cyclic' mode
From this value (50) CANape calculates a prescaler value of 5 and configures a new DAQ list with this prescaler value.

The ECU device calls **ccpDaq(1)** every 10ms in the source code (e.g. in a timer interrupt service routine).

10ms passes: ccpDaq(1) is called; is prescaler == 0 ?, no, so do not send data to CANape, decrement prescaler value to (4)

10ms passes: ccpDaq(1) is called, is prescaler ==0 ?, no, so do not send data to CANape, decrement prescaler value to (3)

10ms passes: ccpDaq(1) is called, is prescaler == 0 ?, no, so do not send data to CANape, decrement prescaler value to (2)

10ms passes: ccpDaq(1) is called, is prescaler == 0 ?, no, so do not send data to CANape, decrement prescaler to (1)

10ms passes: ccpDaq(1), transmit the new measurement values to CANape,
set the prescaler = 5

If the user has configured some measurement signals in “Task1” mode, CANape receives the new measurement values (e.g. Triangle) every 10ms. In “cyclic” mode, CANape receives the new measurement values (e.g. Triangle) every 50ms (if a value of 50 ms has been set in “Rate”).

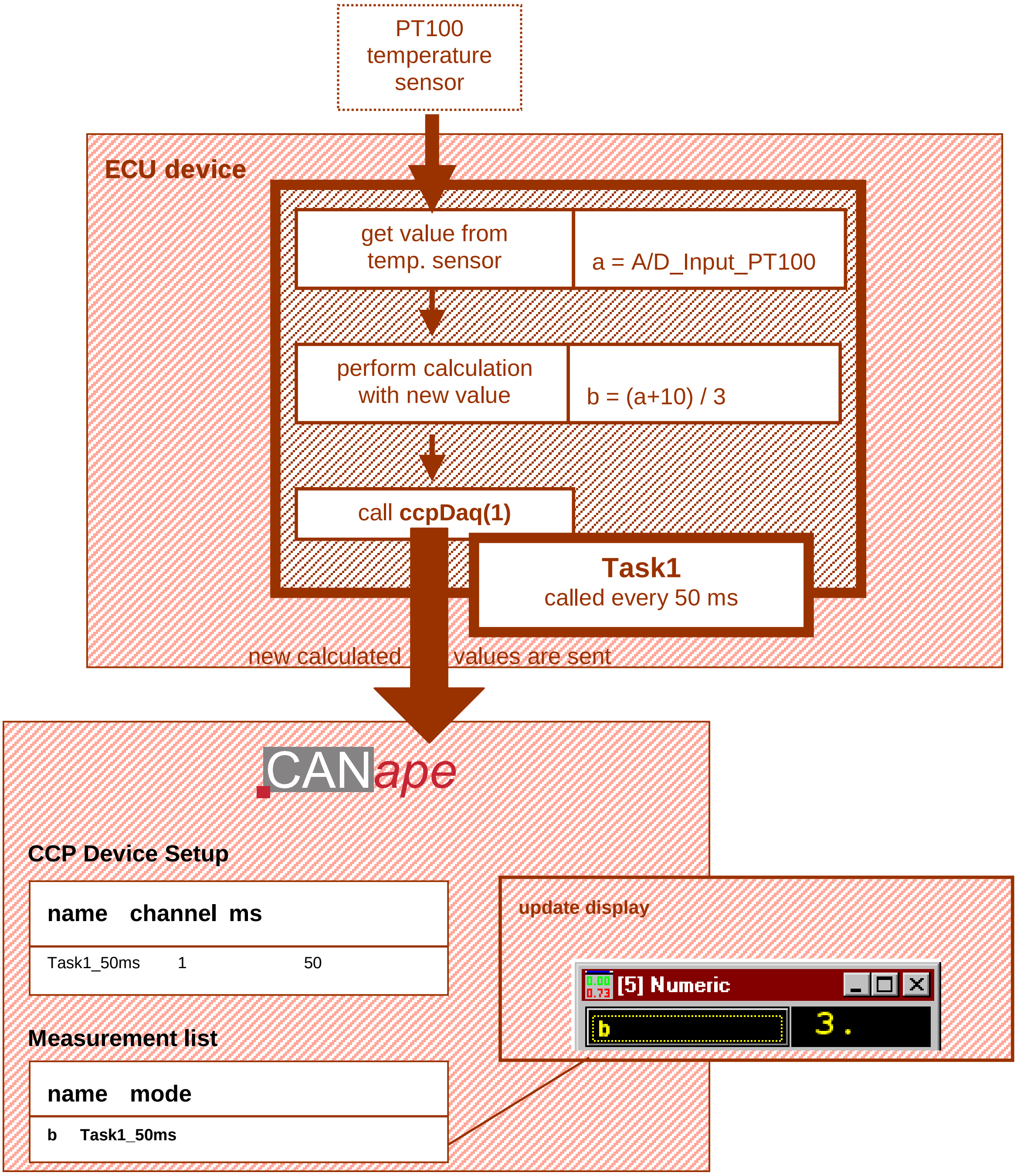
✓ Start the measurement by pressing the F9 key or the ⚡ symbol in the toolbar,

✓ Now a triangle signal should be displayed in the graphic window

3.6.1 Explaining the Synchronous Data Acquisition Mode

This section describes the synchronous data acquisition mode in more detail. In the following example the ECU contains two tasks, one cyclic task, Task1 which is executed every 50 ms and a second task, Task2 which is executed when an external event occurs. The cyclic task reads a sensor value, performs some calculations and calls the *ccpDaq()* function. Each time **ccpDaq(1)** is called by the task, the measurement values are transmitted to CANape (see Figure 14).

Example:



CCP Device Setup

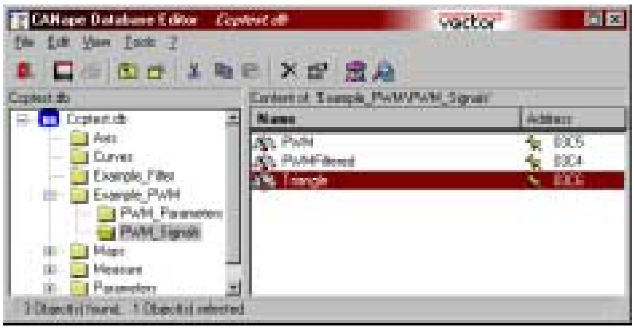
name	channel	ms
Task1_50ms	1	50

Measurement list

name	mode
b	Task1_50ms

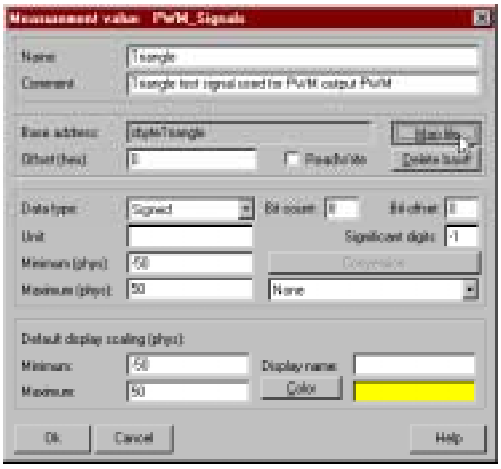
Figure 14: Measuring A Signal From Task1 In Synchronous Data Acquisition Mode

The user defines the variables from the ECU to be measured in the controller database and associates them with the correct address from the linker map file:

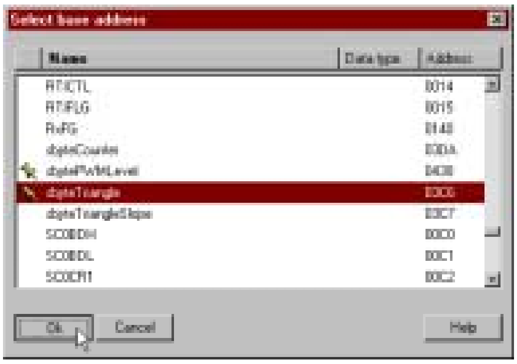


Content of 'Example_PWM/PWM_Signals'

Name	Address
PWM	00C5
PWMFiltered	00C4
Triangle	00C6



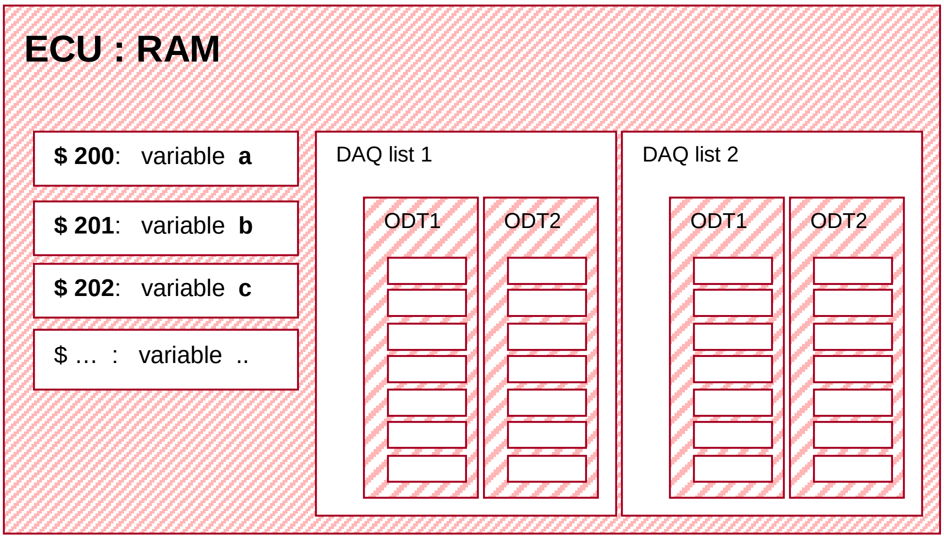
Name: Triangle
Comment: Triangle test signal used for PWM output PWM
Base address: dbyteTriangle
Offset (hex): 0
□ Read/write
Data type: Signed
Bit count: 8
Bit offset: 0
Unit:
Significant digits: -1
Minimum (phys): -50
Maximum (phys): 50
None
Default display scaling (phys):
Minimum: -50
Maximum: 50
Display name:



Name	Data type	Address
RTICTL		0014
RTIFLG		0015
RxFG		0148
dbyteCounter		00DA
dbytePWMLevel		0438
dbyteTriangle		00C6
dbyteTriangleSlope		00C7
SC0BDH		00C0
SC0BDL		00C1
SC0CR1		00C2

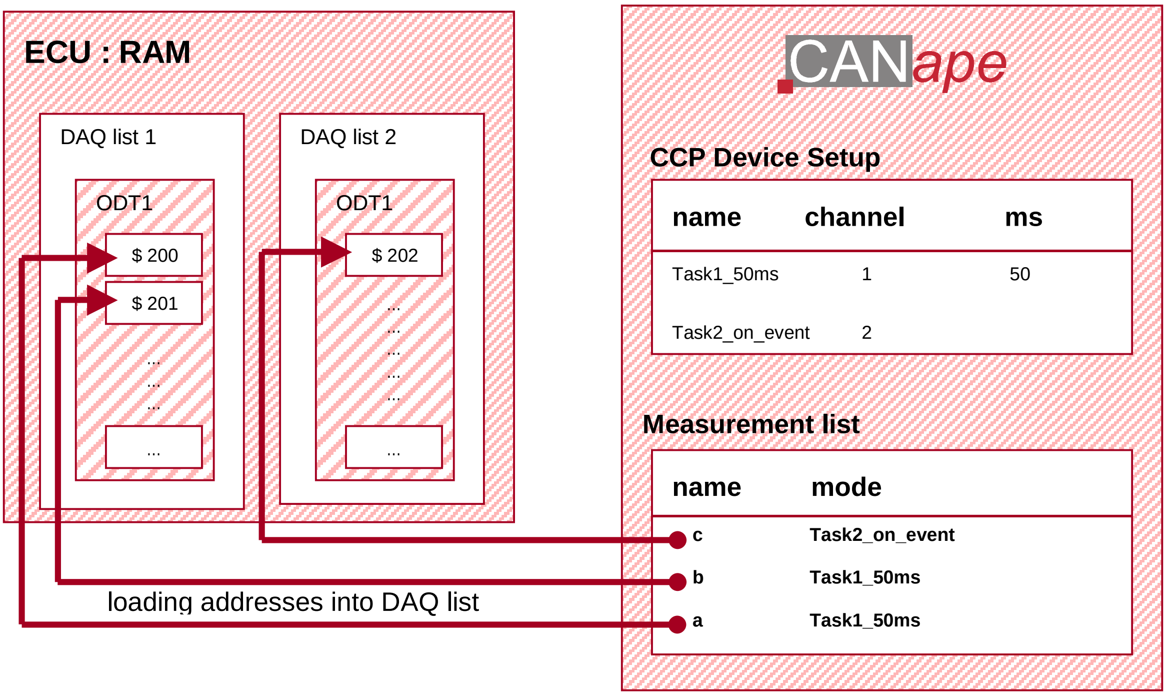
CANape knows the address and data type of each measurement signal stored in the ECU via the controller database. The measurement signals are located in the RAM memory of the ECU. The CCP driver creates as many DAQ lists inside the ECU RAM as defined by CCP_MAX_DAQ in the *ccppar.h* file. A DAQ list contains as many object descriptor tables (ODT) as defined by CCP_MAX_ODT. An object descriptor table is a table which contains up to seven pointers to bytes.

Each ODT is mapped to a CAN DTO message. The first byte contains a packed id, the next seven bytes the measured data values. In the following figure, the ECU contains some measurement signals, like a,b,c etc. and two DAQ lists with two object descriptor tables:



In the ‘Measurement configuration’ dialog the user configures all the measurement signals listed for the current configuration and assigns them a data acquisition mode. In the following example the user wants to measure the signals *a, b* (mode: Task1_50ms) and *c* (mode: Task2_on_event) using synchronous data acquisition.

When the measurement is started by the user via the <F9> key or the ⚡ symbol in the toolbar, CANape configures the DAQ lists via special CCP commands in the ECU’s RAM. This is done by loading the specific address information from the measurement list into the object descriptor tables. In our example, only one of the two object descriptor tables is used by each DAQ list. After the DAQ lists are configured, the CCP driver enables the measurement.



name	channel	ms
Task1_50ms	1	50
Task2_on_event	2	

name	mode
c	Task2_on_event
b	Task1_50ms
a	Task1_50ms

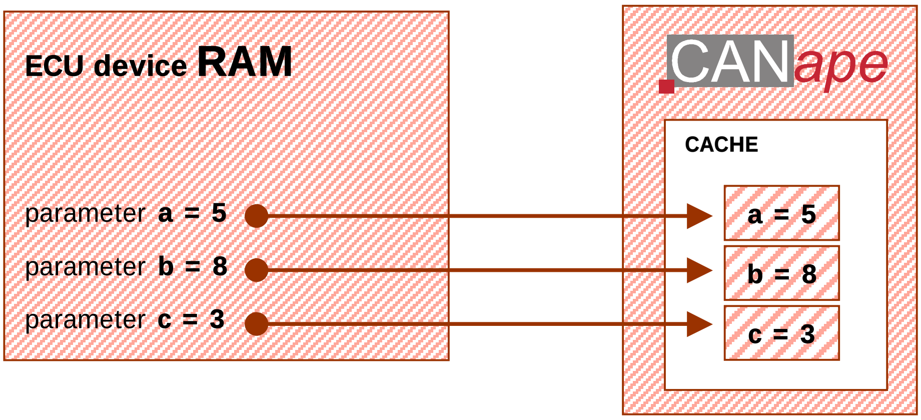
While the measurement is running, the application in the ECU calls **ccpDaq(x)**. The *ccpDaq(x)* generates a CAN message for each ODT of the DAQ list x, stores the created CAN messages in a send queue and starts to transmit the CAN DTO messages to CANape.

3.7 Enabling the Checksum Calculation

In Section 2.3.3 we explained the purpose of the checksum calculation. Without checksum calculation in the ECU, the parameters can only be changed by the user in the online mode. To be able to change parameter values in the offline mode, the checksum calculation in the ECU and the cache memory (also called mirror memory) in CANape have to be enabled.

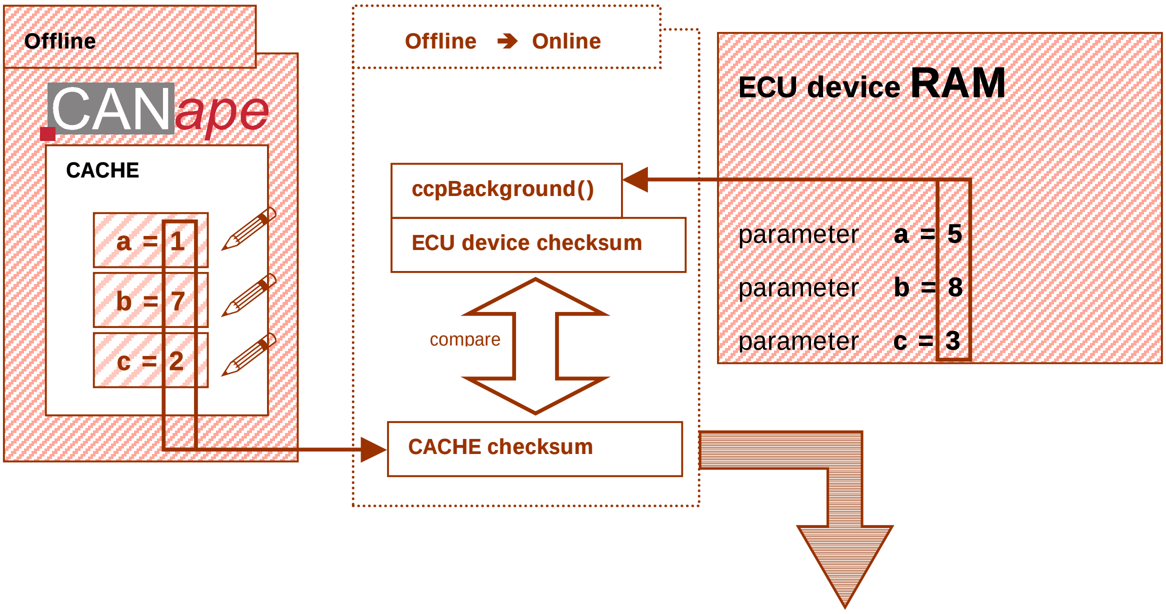
The prerequisite for the checksum calculation is that a few parameters be defined in the ECU. In our example we will use Curve_1 (shared X axis), Curve_2 (shared Y axis) and a Map_8x8 as parameters.

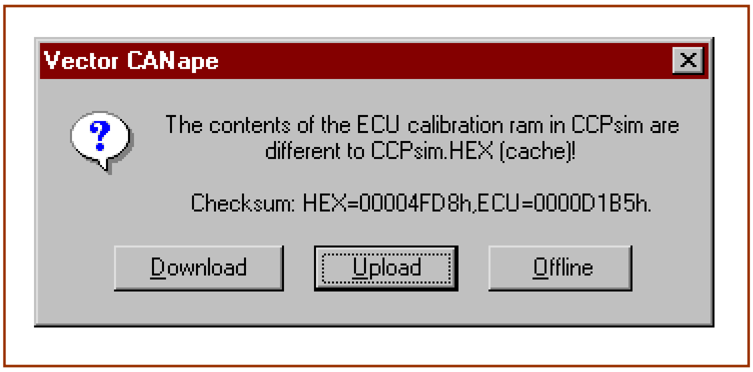
After enabling the cache memory, CANape copies the contents of the calibration RAM of the ECU into its cache:



If the user changes a value of a parameter, the changes are immediately stored in the cache and in the calibration RAM of the ECU. The contents of the cache memory in CANape and the contents of the calibration RAM of the ECU are consistent during the online mode.

If the user works in the offline mode (no connection with the ECU), the values of the parameters can also be changed, but only the values of the cache memory change. When the user changes back to the online mode, CANape compares the checksum of the cache memory with the checksum from the micro controller. The checksum of the ECU is calculated by the **ccpBackground()** function. If both checksums are not equal, CANape opens a message box.





The contents of the ECU calibration ram in CCPsim are different to CCPsim.HEX (cache)!

Checksum: HEX=00004FD8h,ECU=0000D1B5h.

Now the user can choose to upload the values of the parameters from the ECU into CANape or download the current values from the cache memory into the ECU.

3.8 Enabling the Checksum Calculation in the ECU

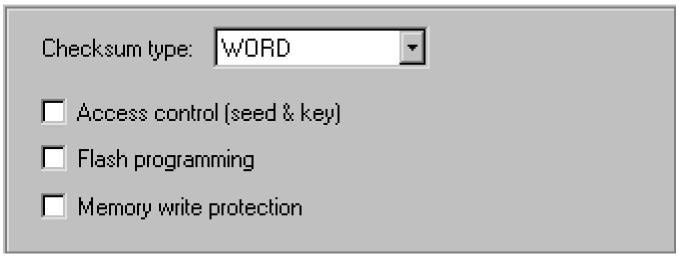
Now we will enable the checksum calculation in the CCP Demo sample. In this example we will use the checksum type WORD. See Chapter 3, *Optional Features of the CCP Driver* in the *CCP.PDF* documentation for more details about the checksum calculation. The *ccpBackground()* function, which calculates the checksum over a memory area, needs to be inserted into your application.

The checksum calculation has to be enabled in the *ccppar.h* file with the CANGen tool:

✓ open CANGen with the previous configuration (in our example ccp_demo.ccf)

✓ click “Generation Options” in the toolbar

✓ change to tab “CCP options” and change “Checksum type” to “WORD”



Checksum type: WORD

☐ Access control (seed & key)

☐ Flash programming

☐ Memory write protection

✓ apply and save all changes, generate the new files

Now follow these steps:

✓ locate a place in your application which is called periodically (e.g. in an idle loop) and add the following line:

```
ccpBackground();
```

✓ save the file

✓ recompile the project and re-program the HC12 with the new code

3.9 Enabling the Checksum Calculation and the Cache in CANape

✓ start CANape via the icon on the desktop

✓ open the CCP Device Setup dialog via “Options **|** Driver configuration...” and select the tab “CCP”

✓ be sure the “Enable” checkbox is checked in the “Checksum” section, and set the “Method” to **ADD_WORD**

✓ change to the “RAM” tab

✓ in this dialog the calibration RAM areas of the ECU can be defined and the cache in CANape can be enabled

✓ insert a new calibration RAM area: enter a start address and the size of the calibration RAM area
This information can be found in the linker map file. In our example the parameters of the ECU are located at:

curve_1_8_uc	$ 3BF	(size = 1 x 8 bytes)
curve_2_8_uc	$ 3C7	(size = 1 x 8 bytes)
map_1_8_8_uc	$ 37F	(size = 8 x 8 bytes)

total size = 80 (**50h**) bytes

CCP DEMO example: start address = 3BFh size = 50h

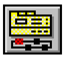
✓ click “Ok”

✓ if all was correctly set you are now able to edit parameters in the offline mode.

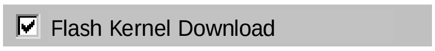
3.10 Enabling “Flash Kernel Download” in the ECU

✓ open CANGen with the previous configuration (in our example ccp_demo.ccf)

✓ click “Generation Options” in the toolbar



✓ change to tab “CCP options” and check “Flash Kernel Download”

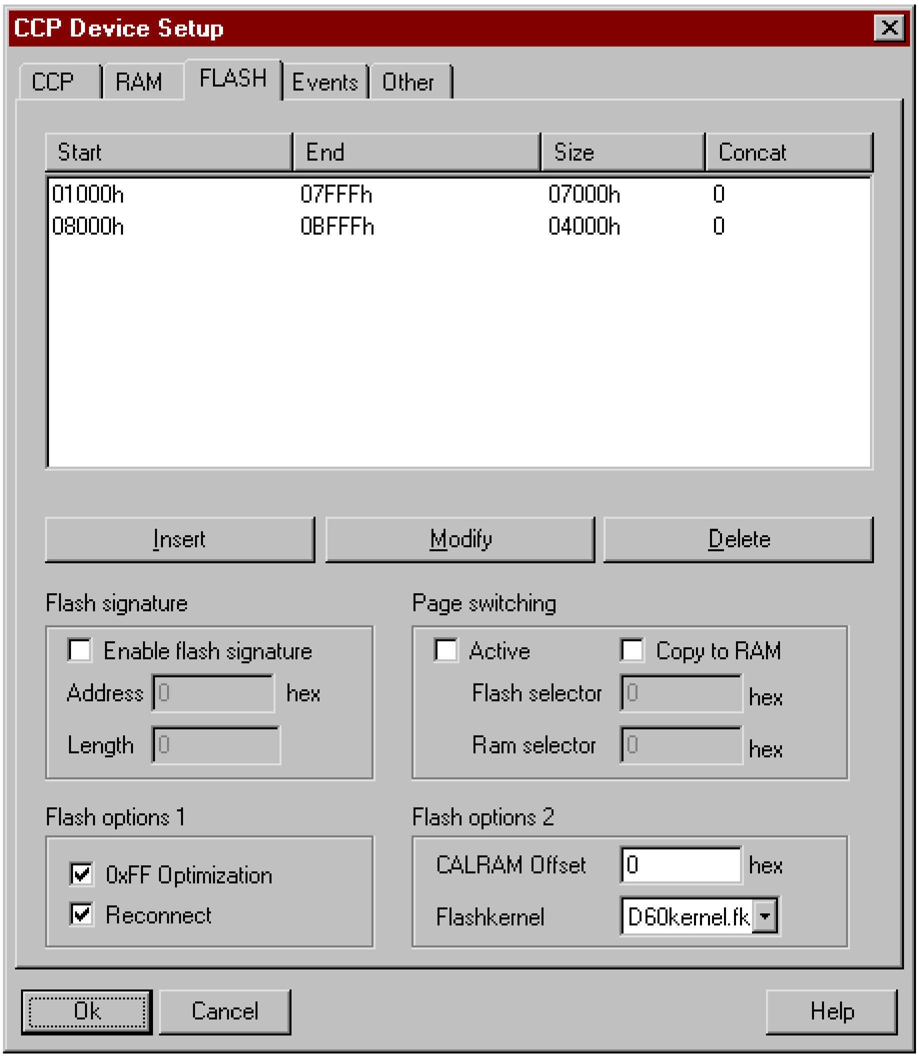


☑ Flash Kernel Download

✓ apply and save all changes, generate the new files

3.11 Enabling “Flash Kernel Download” in CANape

- ✓ open your CANape project, e.g. CANape CCP_DEMO
- ✓ click “Options | Driver configuration…” in the main menu
- ✓ select your device, e.g. HC12 and click “Ok”
- ✓ change to tab “FLASH” and insert flash sections:



Start	End	Size	Concat
01000h	07FFFh	07000h	0
08000h	0BFFFh	04000h	0

Flash signature

☐ Enable flash signature
Address 0 hex
Length 0

Page switching

☐ Active ☐ Copy to RAM
Flash selector 0 hex
Ram selector 0 hex

Flash options 1

☑ 0xFF Optimization
☑ Reconnect

Flash options 2

CALRAM Offset 0 hex
Flashkernel D60kernel.fk

our CARD12.D60 evaluation board has got two flash arrays

Flash Array 1:	$ 1000 - $ 7FFF
Flash Array 2:	$ 8000 - $ FFFF

Flash Array 2 contains the monitor program “TwinPEEK”, so the Flash Array 2 space is minimized to $ 8000 - $ BFFF.

Insert two flash sections:

Flash Array 1:	Start $ 1000	Length 7000h
Flash Array 2:	Start $ 8000	Length 4000h

✓ check “0xFF” Optimization and “Reconnect”

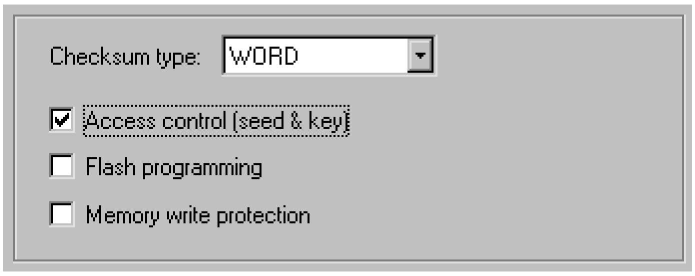
✓ select your Flash Kernel, which is stored in the CANape\Exec folder, e.g. D60flashkernel.fkl

3.12 Enabling “Seed & Key” in the ECU

✓ open CANGen with the previous configuration (in our example ccp_demo.ccf)

✓ click “Generation Options” in the toolbar

✓ change to tab “CCP options” and check “Acces control (seed & key)”



Checksum type: WORD

☑ Access control (seed & key)

□ Flash programming

□ Memory write protection

✓ apply and save all changes, generate the new files

Now follow these steps:

✓ insert the following two functions into the control device application (e.g. in can_ccp.c):

```
ccpGetSeed();

ccpUnlock();
```

```
DWORD ccpGetSeed(BYTE resourceMask) {


  /* Simple example: No algorithm is used here for calculating different seeds! */


  if (resourceMask == PL_CAL) return 22;
  /* Returns the seed for CALIBRATION ACCESS */


  if (resourceMask == PL_DAQ) return 55;
  /* Returns the seed for enabling the DATA AQCUISITION */


  if (resourceMask == PL_PGM) return 99;
  /* Returns the seed for FLASH PROGRAMMING */

  return 0;


}
```

```
BYTE ccpUnlock(BYTE *key) {


  /* Simple example: No algorithm is used here for checking the correct key */



  DWORD test = *((DWORD*)key) ;

  if (test == 22) return PL_CAL;
  /* Check the key for CALIBRATION ACCESS */


  if (test == 55) return PL_DAQ;
  /* Check the key for enabling the DATA AQCUISITION */


  if (test == 99) return PL_PGM;
  /* Check the key for FLASH PROGRAMMING */


  /* Key is not correct -> return 0 */
  return 0;

}
```

- ✓ save the file
- ✓ recompile the project and re-program the HC12 with the new code

Note:
PL_CAL, PL_DAQ and PL_PGM are defined constants in the ccp.h file

3.13 “Seed & Key” – DLL

The “Seed & Key”-DLL is used inside CANape to calculate a key depending on the “Seed”, sent by the ECU.

To create the **seedkey1.dll**, follow these instruction:

✓ Start the Microsoft® development environment,

✓ open project "Seedkey1",

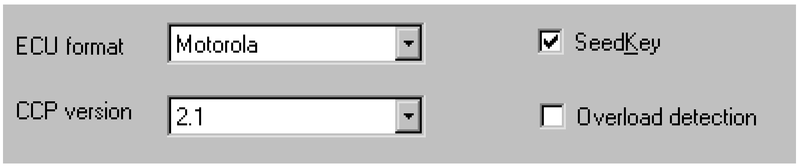
✓ edit the file "seedkey1.cpp" and implement the algorithm for calculating the "key" from the "seed":

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

extern "C" {

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

  {

    /* Implement here the algorithm for calculating the key from the seed */
    /* Here: Simple example: Return the seed value as key */

    *((unsigned long*)key) = *((unsigned long*) seed);
    *sizeKey = 4;
    return TRUE;

  }

}
```

✓ Generate via the Microsoft® -Compiler the file "seedkey1.dll" and copy it into the CANape\Exec directory.

3.14 Enabling “Seed & Key” in CANape

✓ open your CANape project, click “Options | Driver configuration”, select your device and check “SeedKey”



ECU format: Motorola

CCP version: 2.1

☑ SeedKey

☐ Overload detection

Now when the user switches CANape from Offline to Online Mode, he is authorized to start the measurement and to change parameters etc. inside the ECU

3.15 C Code Generation for Calibration Data Objects

This chapter explains a new feature called “C Code Generation” which is integrated in CANape 3.1.50 version or higher.

3.15.1 Template Files

The code generation process is based on templates, which contain macros. All valid macros which are found in a template file are replaced by their values. The target C file always starts with the template file "HEAD.TEMPL" (if available). Furthermore, C include commands, C defines etc. can be located there.

For each calibration object the appropriate template file is determined, the macros are replaced and the result is attached to the end of the target file.

The name of the template file is determined as follows:

If a record layout has been assigned to the calibration object in the database, the name of the record layout is used as template filename with the file extension ".TEMPL". If no record layout exists for the calibration object, the name is determined by the variable type accord to:

Value.TEMPL	scalar parameters
String.TEMPL	ASCII strings
Axis.TEMPL	shared axis
Curve.TEMPL	curves, those x axis points is not stored in the ECU (curves without x axis or with a fixed x axis)
CurveX.TEMPL	curves, those x axis points is stored in the ECU (curves with standard axis or shared axis)
Map.TEMPL	maps, those x- and y-axis points are not stored in the ECU (no x axis or x fixed axis, no y axis or y fixed axis)
MapX.TEMPL	maps, those x- axis points is stored in the ECU and those y- axis points is NOT stored in the ECU x standard axis or x shared axis, no y axis or no y fixed axis)
MapY.TEMPL	maps, those x- axis points is NOT stored in the ECU and those those y- axis points is stored in the ECU (no x axis or no x fixed axis, y standard axis or y fixed axis)
MapXY.TEMPL	maps, those x- and y-axis points are stored in the ECU (x standard axis or x shared axis, y standard axis or y shared axis)

If the template file for a calibration object with record layout, according to the name of the record layout could not be found, then a standard template file according to the variable type will be used.

All the template files must be located in the working directory.

3.15.2 Macros

The following macros are replaced in the template files:

$TEMPLATE_NAME$	File name of the used template files
**$HEADER_COMMENT**$	Comment entered in the dialog
$NAME$	Name of the database variable
$COMMENT$	Comment of the variable from the database
$UNIT$	Physical unit of the variable
$PHYS_MIN$	Physical minimal-value
$PHYS_MAX$	Physical maximal-value
$X_DIMENSION$	Number of x axis points for axis, curves and maps
$Y_DIMENSION$	Number of y axis points for axis, curves and maps
$DATATYPE_C$	Data type in C syntax
$DATATYPE$	Data type in "general formulation" (BYTE,WORD,DWORD,...)
$VARIANT$	Name of the variant for variant coded parameters
$VALUE_LIST$	Value, respectively value list separated by a comma
$MEMORY_DUMP$	Memory dump (List of bytes, separated by a comma)
$INDEX$	Continuous Index
X_AXIS_NAME	Name of the x axis
$X_AXIS_COMMENT$	Comment of the x axis
X_AXIS_UNIT	Physical unit of the x axis
$X_AXIS_PHYS_MIN$	Physical minimal-value of the x axis
$X_AXIS_PHYS_MAX$	Physical maximal-value of the x axis
$X_AXIS_DATATYPE_C$	Data type of the x axis in C syntax
$X_AXIS_DATATYPE$	Data type of the x axis in "general formulation"
$X_AXIS_SOURCE_NAME$	Name of the input source for the x axis
$X_AXIS_SOURCE_COMMENT$	Comment of the input source for the x axis
$X_AXIS_VALUE_LIST$	Value list for the x axis
Y_AXIS_NAME	Comment of the y axis
$Y_AXIS_COMMENT$	Comment of the y axis
Y_AXIS_UNIT	Physical unit of the y axis
$Y_AXIS_PHYS_MIN$	Physical minimal-value of the y axis
$Y_AXIS_PHYS_MAX$	Physical maximal-value of the y axis
$Y_AXIS_DATATYPE_C$	Data type of the y axis in C syntax
$Y_AXIS_DATATYPE$	Data type of the y axis in "general formulation"
$Y_AXIS_SOURCE_NAME$	Name of the input source for the y axis
$Y_AXIS_SOURCE_COMMENT$	Comment of the input source for the y axis
**$Y_AXIS_VALUE_LIST**$	Value list for the x axis

If the value of a macro can not be determined, it will not be replaced (e.g. Y_AXIS_NAME for a curve).

3.15.3 Template Example

Template curve.templ used in CANape

```
/* ---------------------------------------------------------------------------*/
//// Begin "curve.templ" for $NAME$

static $DATATYPE_C$ $NAME$[$X_DIMENSION$] = /* $COMMENT$ */
     { $VALUE_LIST$ };

//// End
```

Generated C Code

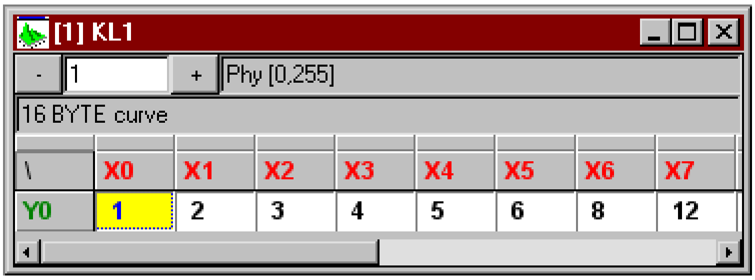
```
/* ---------------------------------------------------------------------------*/
//// Begin "curve.templ" for MY_CURVE

static unsigned char MY_CURVE [8]= /* nothing */
     { 1, 2, 3, 4, 5, 6, 7, 8 };

//// End
```

3.15.4 How to generate a C code file (CCPTEST2.CNA)

Starting point
ecu.c file with an origin curve (named KL2 in CANape):

```
volatile unsigned char curve5_16_uc[16] =
   {1,2,3,4,5,6,8,12,14,11,9,7,6,5,4,3};
```

- ✓ be sure the template **curve.templ** is available in your working directory, see the template example above for details.
- ✓ replace the word ‘static’ with ‘volatile’ in the curve.templ, to generate a correctly set C code for ‘KL2’
- ✓ open CANape project CCPTEST2.CNA
- ✓ select curve **KL1** and change some parameters



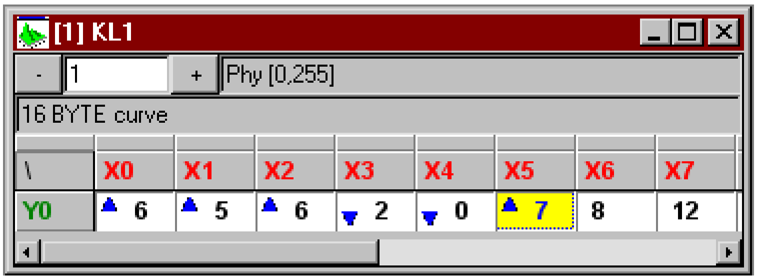
[1] KL1

Phy [0,255]

16 BYTE curve

\	X0	X1	X2	X3	X4	X5	X6	X7
Y0	1	2	3	4	5	6	8	12

Figure 15: original values



[1] KL1

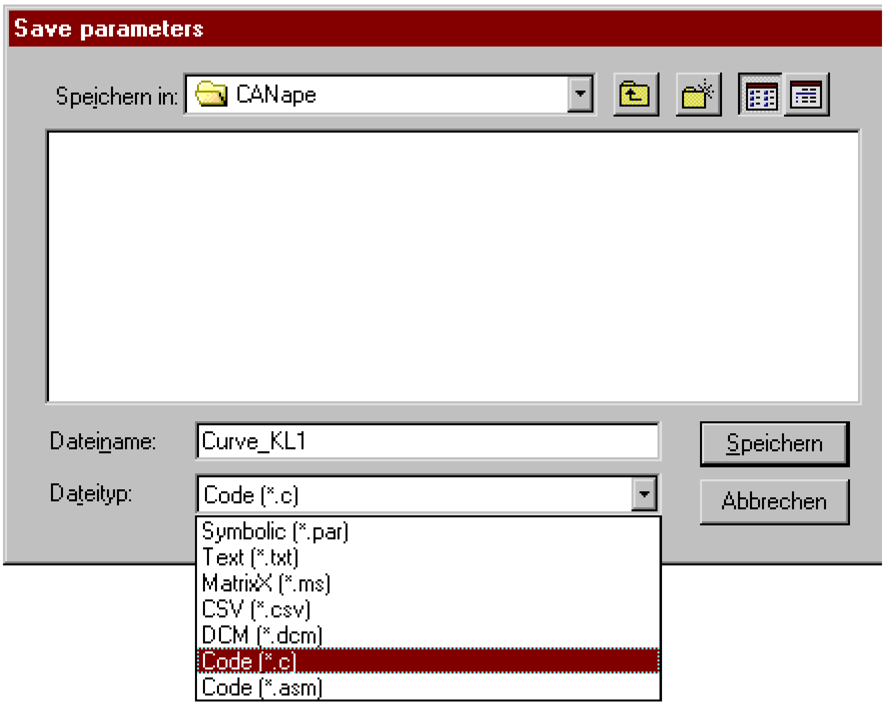
Phy [0,255]

16 BYTE curve

\	X0	X1	X2	X3	X4	X5	X6	X7
Y0	▲ 6	▲ 5	▲ 6	▼ 2	▼ 0	▲ 7	8	12

Figure 16: changed values

- ✓ right-click the curve **KL1**
- ✓ click “Save”, select the file type *.c and enter a file name



Save parameters

Speichern in: CANape

Dateiname: Curve_KL1

Dateityp: Code (*.c)

- ✓ click “Save” and press “Ok”

- ✓ The generated file looks like this:

```
/* ---------------------------------------------------------------------------*
///// Begin "curve.templ" for KL1

volatile unsigned char KL1[16] = /* 16 BYTE curve */

  { 6, 5, 6, 2, 0, 7, 8, 12, 10, 7, 2, 7, 10, 9, 8, 3};

//// End
```

- ✓ to use the generated C code mark/copy the needed sections and replace the origin section of the ecu.c file:

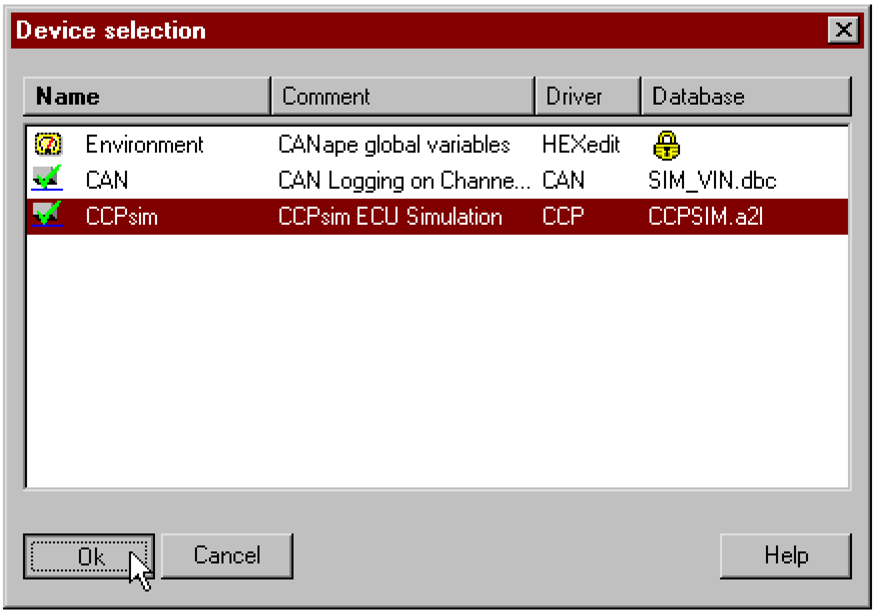
```
…
volatile unsigned char curve4_8_uc[16] =
  {41,42,43,44,45,46,48,52,
  51,52,53,54,55,56,58,62};


volatile unsigned char curve5_16_uc[16] =
{1,2,3,4,5,6,8,12,14,11,9,7,6,5,4,3};            / replace old parameters  with….


volatile unsigned char curve5_16_uc1[16] =
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
…
…
```

```
…
…
volatile unsigned char curve4_8_uc[16] =
  {41,42,43,44,45,46,48,52,
  51,52,53,54,55,56,58,62};


volatile unsigned char curve5_16_uc[16] =
{ 6, 5, 6, 2, 0, 7, 8, 12, 10, 7, 2, 7, 10, 9, 8, 3};   // …..new ones

volatile unsigned char curve5_16_uc1[16] =
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
…
…
```

To generate a C code file with all parameters you have to:

- ✓ click “Calibration | Save parameterset as…” in the main menu

- ✓ select your device



Name	Comment	Driver	Database
Environment	CANape global variables	HEXedit	
CAN	CAN Logging on Channe...	CAN	SIM_VIN.dbc
CCPsim	CCPsim ECU Simulation	CCP	CCPSIM.a2l

- ✓ click “Ok””, select the file type *.c and enter a file name
- ✓ click “Save” and press “Ok”

4 CCP_DEMO Sample Files

SOURCE FILES

ccp_demo.c

main application,
initializes timer
initializes CAN driver
initializes Vector CCP Driver

can_drv.c

VECTOR CAN Driver

can_ccp.c

interface for CAN and CCP driver

ccp.c

VECTOR CCP Driver

ecu.c

ECU simulation with some parameters etc.

hc12.c

generated by CANGen, contains data buffer for receive/send objects

timer.c

Timer_0. Calls ccpDaq(1) frequently by timer interrupt

vectable.c

contains vector table (for CARD12.D60 demo board only!)