Integration of the Vector CCP Driver with a free CAN Driver

Version 1.1.4
06/10/03

Application Note AN-IMC-1-001

Author(s) Ko,Iz

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

Revision history

Version	Author	Description
1.1.4	Iz	Minor changes
1.1.3	Ko	Seed & Key / Flash Kernel description changed
1.1.2	Ko	Some emphasis changed/added
1.1.1	Ko, Iz	Some details added, Iz released the document
1.1	Ko	Chapter “C Code generation” added
1.0	Ko, Iz	Documentation started

Table of Contents

Contact Information: www.vector-informatik.de or +49 (0)711 80670-0

1.0 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 CAN Driver. A free CAN Driver is provided with the CCP Test demo example. Detailed information about integrating the CCP Driver step by step into a Motorola HC12 microcontroller is described in later chapters with the CCP Test demo sample.

Note: CANape Graph V4.0 has been used for doing the screen shoots in this application note.

The Vector CCP Driver is delivered with three files:

- ccp.c : Vector CCP Driver Source Code
- ccp.h : Header file for the Vector CCP Driver
- ccppar.h : File to configure the Vector CCP Driver

The Vector CCP Driver is configured by editing the *ccppar.h* file, which 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 1. The application calls the CCP Driver function *ccpInit()* once, e.g. after the initialization of the CAN Driver. The CAN Driver which is part of the CCP_TEST demo example is delivered for Motorola HC12 microcontrollers, which works in *polling mode*. This means that the receive buffer of the microcontroller has to be polled by the application cyclically. If a new CRO message arrives, the Vector CCP Driver function *ccpCommand()* has to be called (see Figure 1).

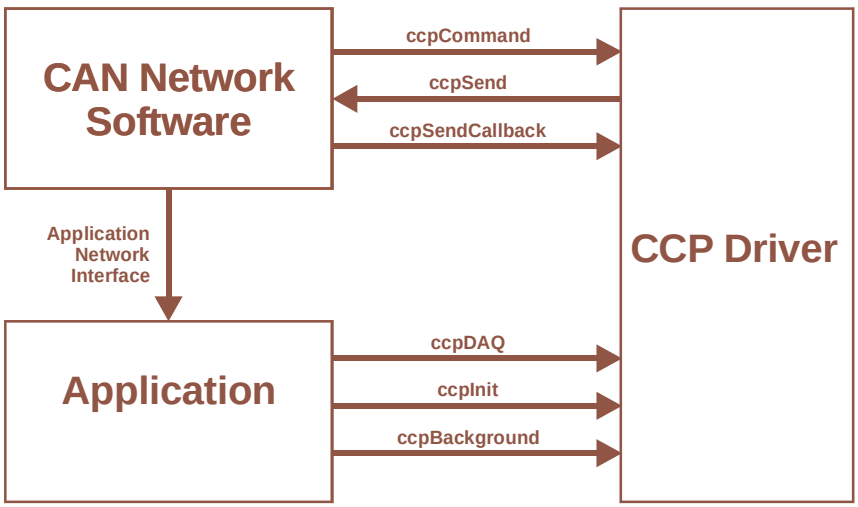


Figure 1: Functional Interface Overview Diagram

The CCP Driver uses the *ccpSend()* function to send DTO messages to the measurement and calibration system, e.g. CANape Graph. 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 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 0). If the communication between CANape Graph and the ECU is set up properly, the other CCP options, like the synchronous data acquisition mode, the checksum calculation function can be enabled in further steps (see Section 2.5)

1.1 Using the CAN Driver

In this chapter the functionality of the provided CAN Driver is introduced. This CAN Driver is operated in the polling mode. The header file *boot_can.h* has to be included in the main application.

The CAN Driver contains the following functions:

- ccpBootInit() : Initializes the CAN controller
- ccpBootTransmitCrmPossible() : Checks to see if a transmission is possible
- ccpBootTransmitCrm() : Transmits a CAN message
- ccpBootReceiveCro() : Receives a CAN message

The following must be done to **send** a single CAN message with a user-defined ID:

- ✓ a send buffer (unsigned char array) of eight bytes must be defined
- ✓ the send buffer must be initialized
- ✓ ccpBootInit (receive_ID, transmit_ID) must be called in the application with a CAN receive and a transmit ID in hex
- ✓ ccpTransmitCrm(send_buffer) must be called

Example:

```
…
…
…
#include "boot_can.h"
…
unsigned char send_buffer [8] = {1, 2, 3, 4, 5, 6, 7, 8};
…
…
…
void main(void) {
  ccpBootInit(0x100,0x101);              // CAN receive ID =100 h, CAN transmit ID = 101h
  …
  …
  …
  …
  ccpBootTransmitCrm(send_buffer);       // send data frame
  …
  …
};
```

The following must be done to **receive** a single CAN message with a user-defined ID:

- ✓ a receive buffer (unsigned char array) of eight bytes must be defined
- ✓ ccpBootInit (receive_ID, transmit_ID) must be called in the application with a CAN receive and a transmit ID in hex
- ✓ ccpReceiveCro(receive_buffer) must be called

Example:

```
…
…
…
#include "boot_can.h"
…
unsigned char receive_buffer [8];
…
…
…
void main(void) {
  ccpBootInit(0x100,0x101);              // receive ID = 100 h, transmit ID = 101h
  …
  …
  …
  …
  ccpBootReceiveCro(receive_buffer);     // receive data frame
  …
  …
};
```

1.2 Configuring the CCP Driver

The *ccppar.h* file is the most important configuration file for the Vector CCP Driver and must be adjusted by the user for the specific ECU. This file contains many defines like:

```
  …
  …
  /*-----------------------------------------------------------------------------*/
  /* CCP parameters */

  /* CCP Identifiers and Address */
  #define CCP_STATION_ADDR  0x0039      /* Define CCP_STATION_ADDR in Intel Format */

  #define CCP_STATION_ID    "ECU00001"  /* Plug&Play station identification */

  #define CCP_DTO_ID        2017        /* CAN identifier ECU -> Master */
  #define CCP_CRO_ID        1639        /* CAN identifier Master -> ECU */
```

```
/*----------------------------------------------------------------------------*/
/* CCP Data Acquisition Parameters */

;#define CCP_DAQ               /* Enable synchronous data acquisition in ccpDaq() */
;#define CCP_MAX_ODT 3         /* Number of ODTs in each DAQ list */
;#define CCP_MAX_DAQ 2         /* Number of DAQ lists */
…
…
```

All the configuration options to be enabled or disabled by the defines are described in the *CCP.PDF* documentation provided with the Vector CCP Driver.

For a basic CCP Driver implementation only

CCP_STATION_ADDR, **CCP_STATION_ID**, **CCP_DTO_ID** and **CCP_CRO_ID**

have to be defined.

In Section 2.1 the *ccppar.h* file for the CCP Test demo example will be configured in detail.

1.3 Writing the Interface Functions for the CCP Driver

These functions are described in Section 2.3, *Description of Interface Functions* of the *CCP.PDF* documentation provided with the Vector CCP Driver.

The functions needed when using the provided CAN Driver are *ccpSend(), ccpGetPointer()* and *ccpUserBackground()*.

In the following an excerpt from the delivered *ccp_can_interface.c* file is shown:

```
// ------------------------------------------------------------------------
// SENDING an CRM-DTO when receiving an CRO
// ------------------------------------------------------------------------
BYTE ccpSend( BYTEPTR msg )
{
  ccpBootTransmitCrm(msg);
                                      // call CAN-send
  return 1;
}
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
// CONVERT pointer
// ------------------------------------------------------------------------
MTABYTEPTR ccpGetPointer( BYTE addr_ext, DWORD addr )        // get Pointer into
{                                                            // normal C
  return (MTABYTEPTR) addr;
}
// ------------------------------------------------------------------------


// ------------------------------------------------------------------------
// PERFORM additional calculations
// ------------------------------------------------------------------------
void ccpUserBackground( void )
{
```

```
}
// -----------------------------------------------------------------------------
```

1.4 Integration of the Vector CCP Driver into an ECU

The following files also have to be included with the application:

- **ccp.c** : Vector CCP Driver
- **ccp.h** : Header file of the Vector CCP Driver
- **ccppar.h** : Header file which configures the CCP Driver
- **ccp_can_interface.c** : Interface functions for the CCP Driver
- **ccp_can_interface.h** : Header file of the interface functions for the CCP Driver

The following steps below are needed to integrate the CCP Driver:

- ✓ the ccppar.h file must be opened with an editor to configure the basic CCP Driver
- ✓ ccp.h, ccppar.h, ccp_can_interface.h must be included in your application
- ✓ a receive buffer (unsigned char array) of eight bytes must be defined, like `unsigned char receive_buffer [8];`
- ✓ ccpBootInit(CCP_CRO_ID, CCP_DTO_ID) must be called in the application (initializes the CAN Driver)
- ✓ ccpInit() must be called (initializes the CCP Driver)
- ✓ check if a new CRO message in receive buffer is available, if so ccpCommand(receive_buffer) must be called
- ✓ all interrupts must be disabled (protect ccpSendCallBack() function call)
- ✓ check if data transmission is possible
- ✓ ccpSendCallBack() must be called
- ✓ all interrupts must be enabled
- ✓ the instructions for the compiler and linker for the ccp.c and ccp_can_interface.c files must be added

The integration of the CCP Driver in the ECU source code could look like in this example:

```
void main(void) {

   ccpBootInit(CCP_CRO_ID, CCP_DTO_ID);          // initialize CAN Driver
   ccpInit();                                    // initialize CCP Driver

   while(1) {
     if (ccpBootReceiveCro(receive_buffer)) {    // if receive buffer full,
       ccpCommand(receive_buffer);               // call ccpCommand
     }
     CCP_DISABLE_INTERRUPT;                      // Disable Interrupts
     if (ccpBootTransmitCrmPossible()) {         // if new transmission is possible
       ccpSendCallBack();                        // call SendCallBack
     }
     CCP_ENABLE_INTERRUPT;                       // Enable Interrupts
   }
}
```

1.5 Enabling the 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.

The polling mode has to be selected in the CANape Graph measurement configuration list. The user has to assign a data acquisition rate for each measurement signal (see Figure 2).

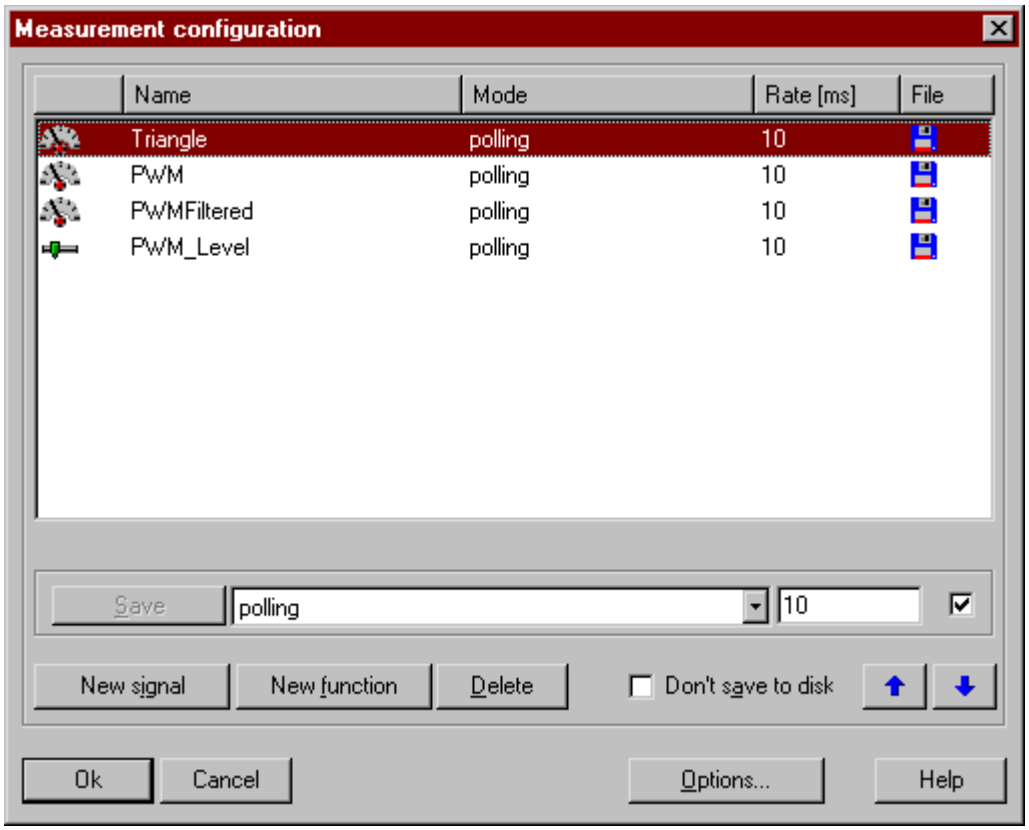


Figure 2: Configuration of the measurement list

CANape Graph 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 2.4 will be showed how the polling data acquisition mode of the HC12 demo example can be enabled.

1.6 Enabling the Synchronous Data Acquisition Mode (DAQ)

The synchronous data acquisition mode has to be enabled in the *ccppar.h* file. The needed defines CCP_DAQ, CCP_MAX_ODT, CCP_MAX_DAQ, which must be defined, and the optional defines CCP_SEND_QUEUE and CCP_SEND_QUEUE_OVERRUN_INDICATION which can be enabled, are described in Chapter 3, *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 Graph measurement list are sent automatically to CANape Graph during the ccpDaq(x) call. The number “x” represents 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 Graph. 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_10ms* because these signals are used in Task 1. These signals will be transmitted automatically from the ECU to CANape Graph every 10ms, 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 the signals u,v and w will be transmitted automatically by the ECU to CANape Graph.

Example of an ECU program with two tasks:

ECU device

main.c

```
…
canInit ();
ccpInit();
…
activate every 10ms 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);
```

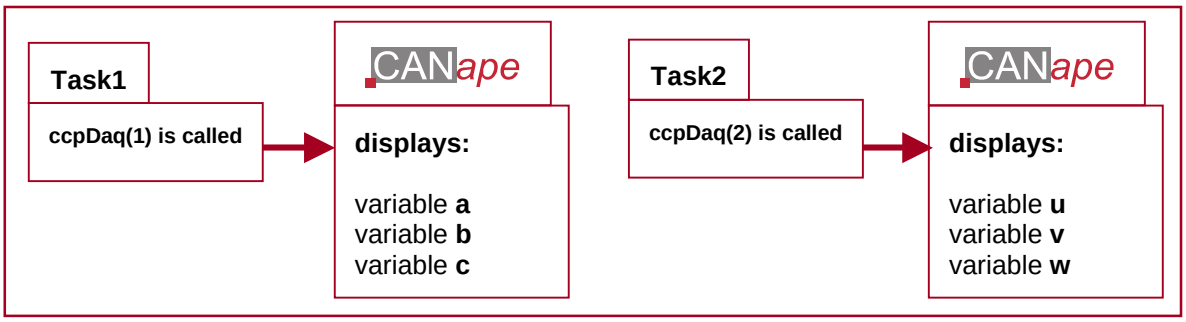
CANape

ce Setup (“Events” tab)

name	channel	ms
Task1_10ms	1	10
Task2_on_event	2	0

name	mode
a	Task1_10ms
b	Task1_10ms
c	Task1_10ms
u	Task2_on_event
v	Task2_on_event
w	Task2_on_event

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.



In Section 2.5 we will see in detail how to enable the synchronous data acquisition mode in our HC12 demo sample.

1.7 Checksum Calculation

The use of a cache in CANape Graph 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 Graph is that the calibration engineer can also change parameters if the ECU is offline.

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

In order to use the cache in CANape Graph, 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. The needed defines, CCP_CHECKSUM and CCP_CHECKSUM_TYPE, which have to be enabled are described in Chapter 3, *Optional Features of the CCP Driver* in the *CCP.PDF* documentation provided with the Vector CCP Driver.

In Section 2.6 we will see in detail how to enable the checksum calculation in our HC12 demo sample and how to enable the cache in CANape Graph.

1.8 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 Graph. 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 Graph 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 Graph.

In chapter 2.7 we will show how to enable the "Flash Kernel Download" in our HC12 CCP_TEST example.

1.9 Seed & Key

The development 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 in the *ccppar.h* file.

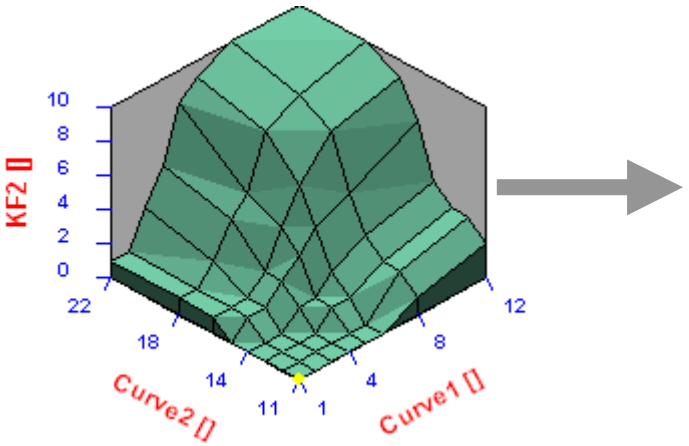
A seed is sent by the ECU to CANape Graph when a connection occurs (switching from Offline to Online Mode). CANape Graph has to calculate the correct key with an special *.dll (see section 2.8) and return it to the ECU. Finally the ECU grants access if the key is correct.

In section 2.8 we will see in detail how to enable "Seed & Key" in our HC12 demo sample.

1.10 C Code Generation of Parameters

The C code generation is a new feature starting with CANape Graph 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 Graph, and the user whishes consistent data in his C source code.

This is an example of “CCP Test” with the *ccptest1.cna* configuration file. The map shown below is stored in the *ecu.c* source.




```
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,  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
  } ;
```

In Section 2.9 we will see in detail how to generate a C code file with the HC12 demo example.

2.0 CCP_TEST Demo example

In this section it will be demonstrated how to interface the Vector CCP Driver to an existing CAN Driver using the CCP Test demo example. The CCP Driver will be integrated into a Motorola HC12 microcontroller and step by step the different CCP options will be, like the synchronous data acquisition mode and the checksum calculation. At the same time CANape Graph will be configured to make measurements in the polling mode, in the synchronous data acquisition mode, and the cache will be enabled.

All the needed files (e.g. the source code files, CANape Graph project for the CCP Test demo sample) are provided with the CCP Test demo example.

The CCP_TEST demo sample simulates an ECU program. The demo example contains some parameters, curves and maps which can be changed by CANape Graph. Some randomized variables, and calculated variables which depend on parameters, are also included in the demo sample. As shown in figure 3, Task 1 is called cyclically all 10 ms from the ECU. Every time Task 1 runs 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 Graph, the effect can immediately be visualized in CANape Graph.

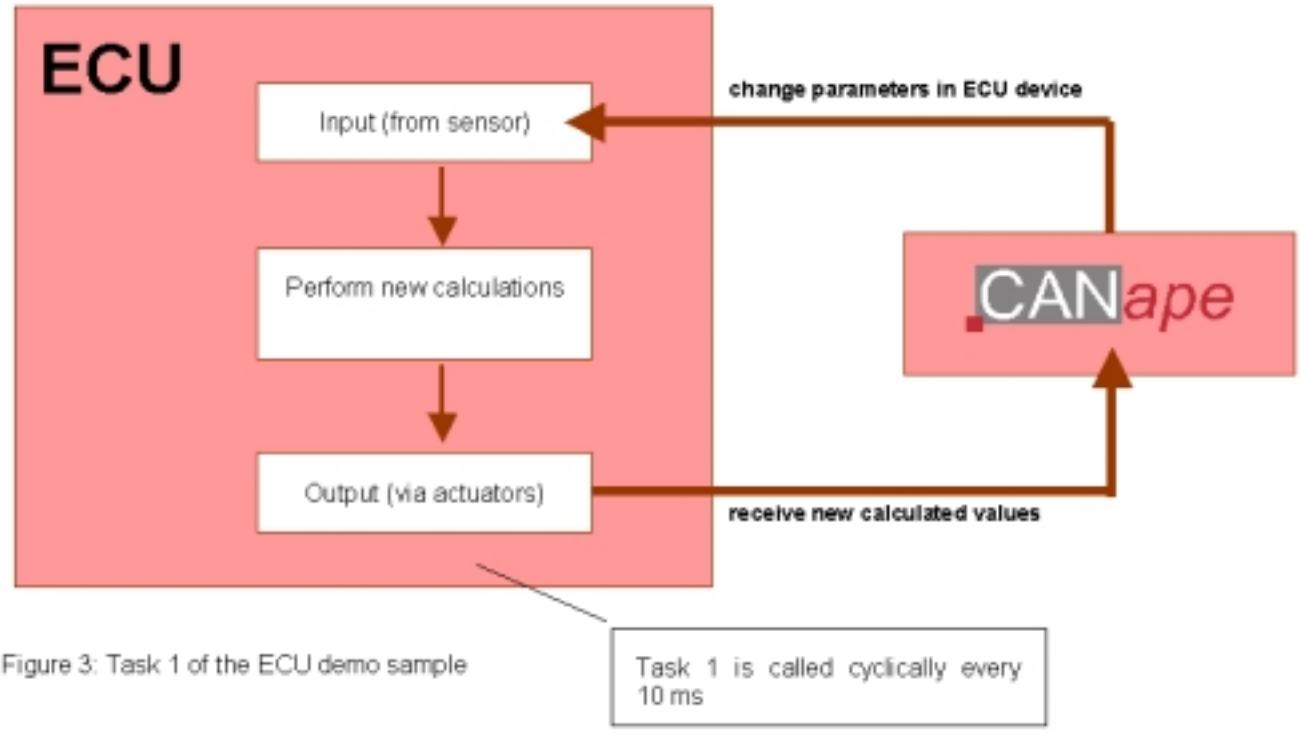


Figure 3: Task 1 of the ECU demo sample

Figure 3 briefly describes the source code files of the CCP Test demo example.

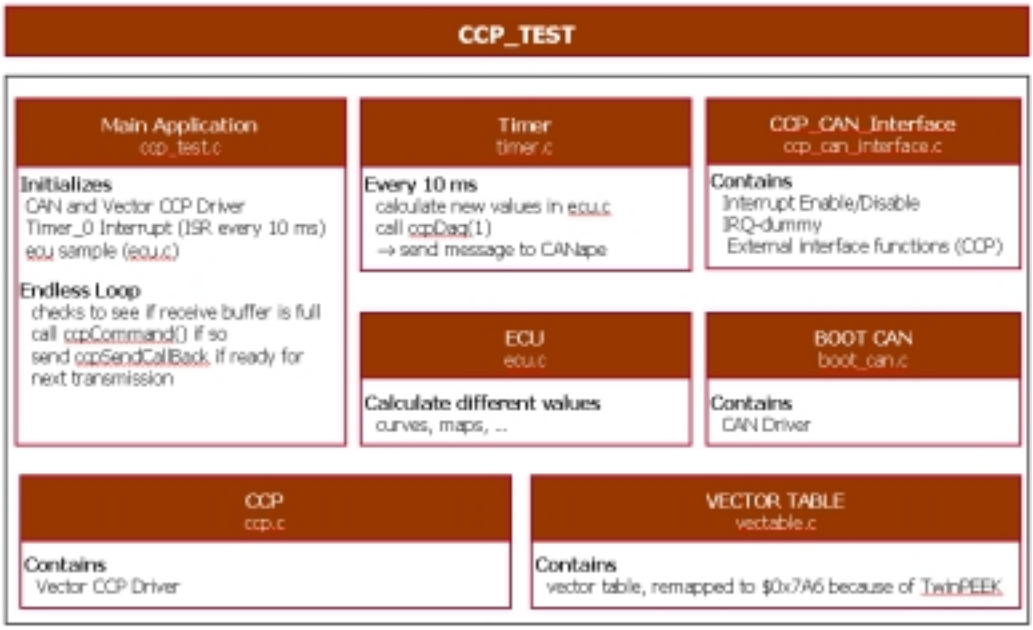


Figure 3: CCP Test demo example - Source Code Structure

2.1 Integration of the Basic CCP Driver in the ECU

The steps discussed in section 1.4 will be done now to integrate the CCP Driver into the ECU.

In the CCP TEST demo sample the basic CCP Driver has to be configured via the *ccppar.h* file with the following settings:

```
...
/*----------------------------------------------------------------------------*/
/* Disable/Enable Interrupts */

/* Has to be defined if ccpSendCallBack will interrupt ccpDaq */

extern void disable_interrupt(void);
extern void enable_interrupt(void);

#define CCP_DISABLE_INTERRUPT disable_interrupt();
#define CCP_ENABLE_INTERRUPT  enable_interrupt();

/*----------------------------------------------------------------------------*/

/* CCP parameters */

/* CCP Identifiers and Address */
#define CCP_STATION_ADDR  0x0000

#define CCP_STATION_ID    "CCPtest"  /* Plug&Play station identification */

#define CCP_DTO_ID        0x101         /* CAN identifier ECU -> Master */
#define CCP_CRO_ID        0x100         /* CAN identifier Master -> ECU */
…
…
```

Next, the *ccp.c* source code files and the *can_ccp_interface.c* file have to be included in the ECU application.

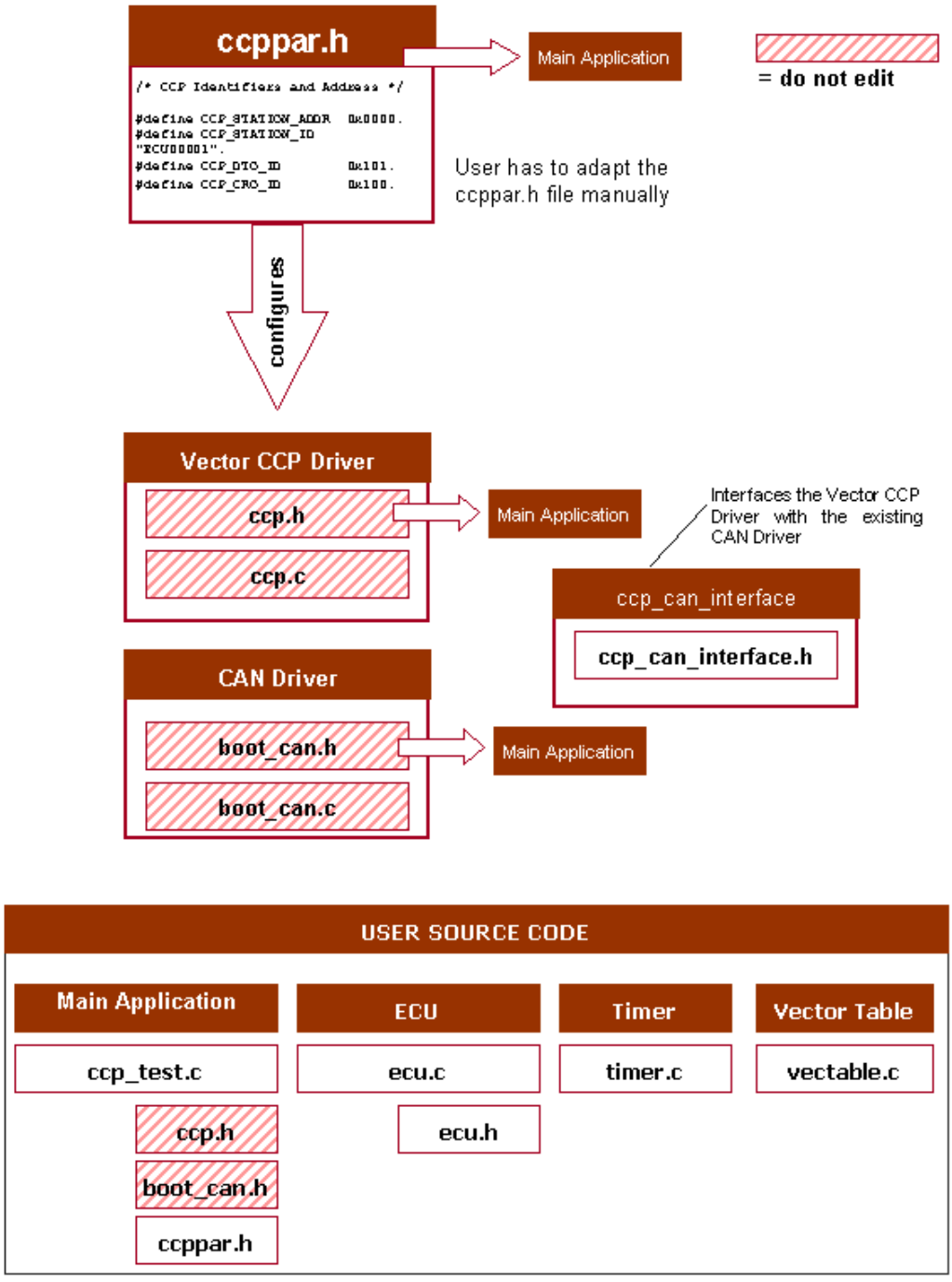


Figure 4: Dependencies of the files used in the CCP_Test demo sample

In the file *ccp_test.c* the *ccpInit()* function has to be added to initialize the CCP Driver. The receive buffer of the HC12 is checked in an infinite loop for new CRO messages from CANape Graph. When a new CRO CAN

message is received, the function *ccpCommand()* is called. The ccpCommand() interprets the contents of the CRO CAN message and responds with a DTO CAN message.

This completes the integration of the basic CCP Driver in the ECU. Next a new CANape Graph project will be created and then a measurement configuration to acquire measurement data in the polling mode will be configured.

2.2 Creation of a new CANape Graph Project

In this chapter a new CANape Graph project for the CCP Test demo sample will be created. Then CANape Graph will be started and the HC12 with the CCP Test demo sample will be added in the CANape Graph device list.

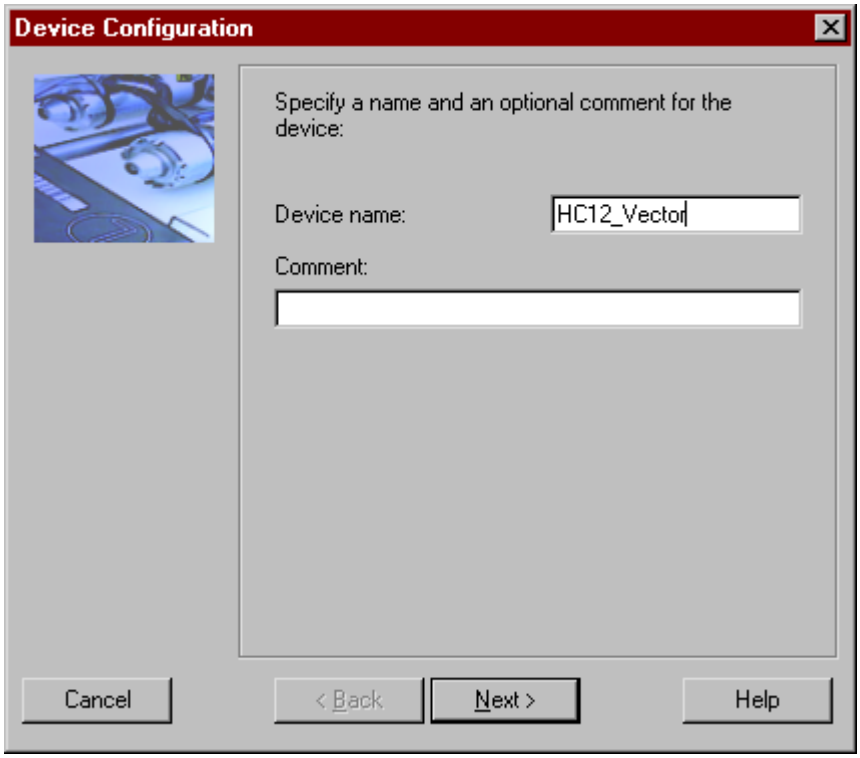
- ✓ "**Create new project"** in the windows start menu in the CANape Graph program folder must be choosen
- ✓ click **<Next>** and enter a name for your project, e.g. CCP_TEST
- ✓ in the next window a path for your new project must be selected
- ✓ be sure the checkbox **"Place shortcut on Desktop"** is checked;
  uncheck **"Start menu"** if necessary
- ✓ click **<Next>**. Do NOT **"Start CANape immediately"**, but click instead **<Finish>**
- ✓ on the Windows desktop a new icon with the name of your project has been created:

- ✓ to start CANape Graph double click this icon

In the next step, a new device (ECU) has to be added in the CANape Graph device list:

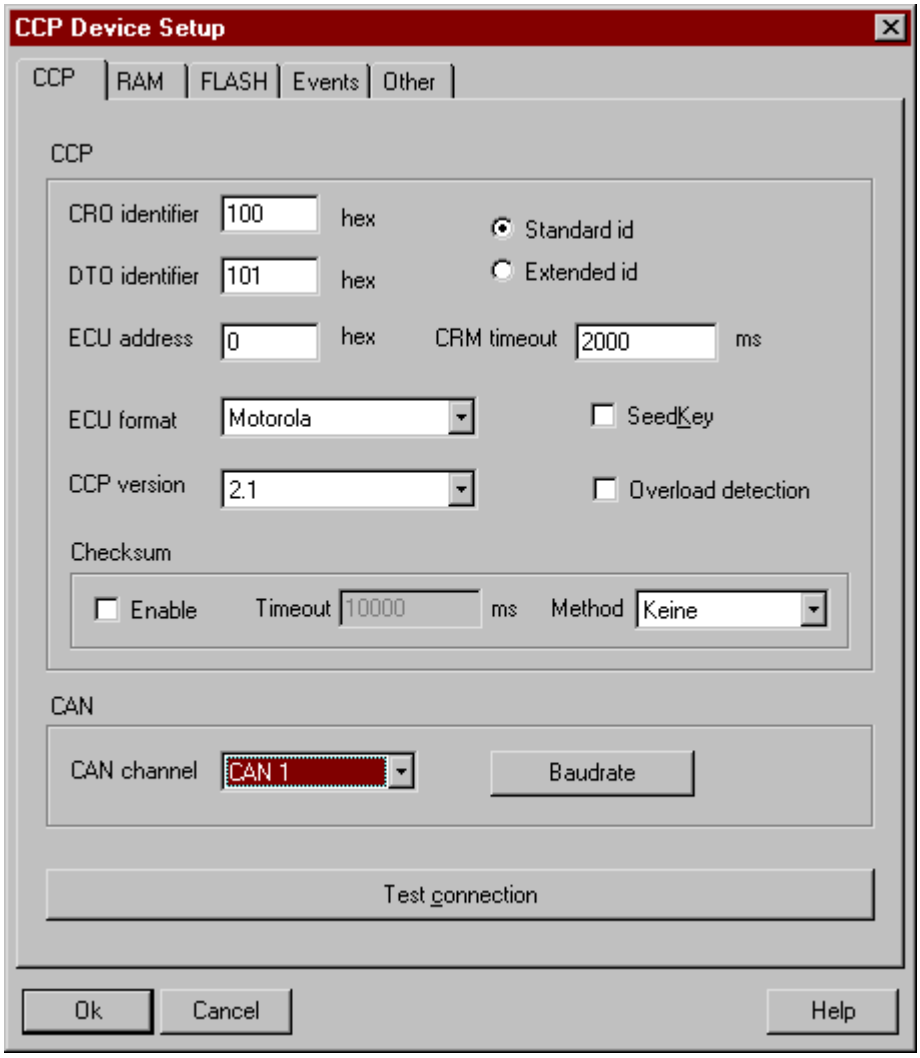
- ✓ press on the  symbol in the CANape Graph toolbar or select the menu command **Tools | Device configuration** to add a new device in the CANape Graph device list
- ✓ press the **<New>** button

- ✓ enter a name for your device, e.g. HC12_Vector



- ✓ click **<Next>**
- ✓ select the type of the driver used in your device (here: CCP).
  Select the CCP driver, press the tab key to enter the drivers options and click on the **<Driver configuration>** button
- ✓ CRO-Identifier: enter the ID for the CCP driver (in this example: 100h)
- ✓ DTO-Identifier: enter the ID for the CCP driver (in this example: 101h)
- ✓ ECU-Address: enter the ECU address (in our example: 0)
- ✓ ECU-Format: Motorola

- ✓ uncheck **"Seed+Key"**, "**Overload detection**" and **"Enable"** from the Checksum area. The tab page should now look like this:



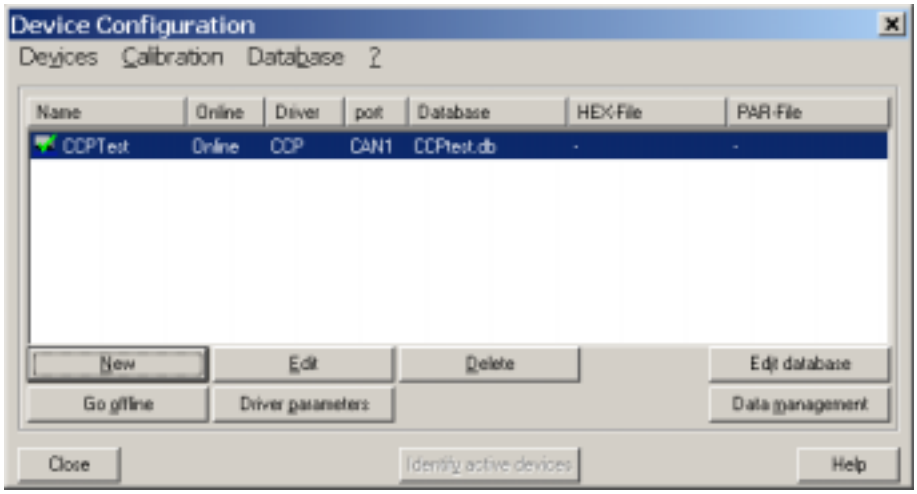
- ✓ the CAN channel and the baud rate must be adjusted (e.g. CAN1, 500kBit/s)
- ✓ click **<Ok>**
- ✓ exit the dialog with **<Ok>** and click on the **<Next>** button
- ✓ the default settings on the next dialog page can be used, click on the **<Next>** button
- ✓ the entry **"one map file"** has to be selected and select the map file directory with the **<Browse>** button
- ✓ map file format: select the COSMIC format
  (our CCP Test demo sample was compiled with the COSMIC C compiler)

- ✓ enter the name of the map file **"ccp_test"**. The extension is automatically set (the default extension name is .MAP)

  Note: A linker map file is needed 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 linker map format, also the data type.

  - ✓ click **<Next>**
  - ✓ The device configuration dialog displays now a window asking to specify the directory for the device's hex file. Accept the default settings. Click on the **<Next>** button

- ✓ The Project Wizard displays now a summary of your settings. Click on the **<Back>** button to go back to previous pages to correct an entry, or click **<Ok>** if all the entries are ok
- ✓ CANape Graph tries now to get a connection to the ECU and opens a dialog to ask if a database should be imported. Click **<No>**.
- ✓ Because no ASAP2 database exists, CANape Graph also asks if an empty ASAP2 database should be created. **<Yes>** must be selected here.
- ✓ If CANape Graph could go online with the ECU, a new created device is displayed with a green check symbol in the device configuration list

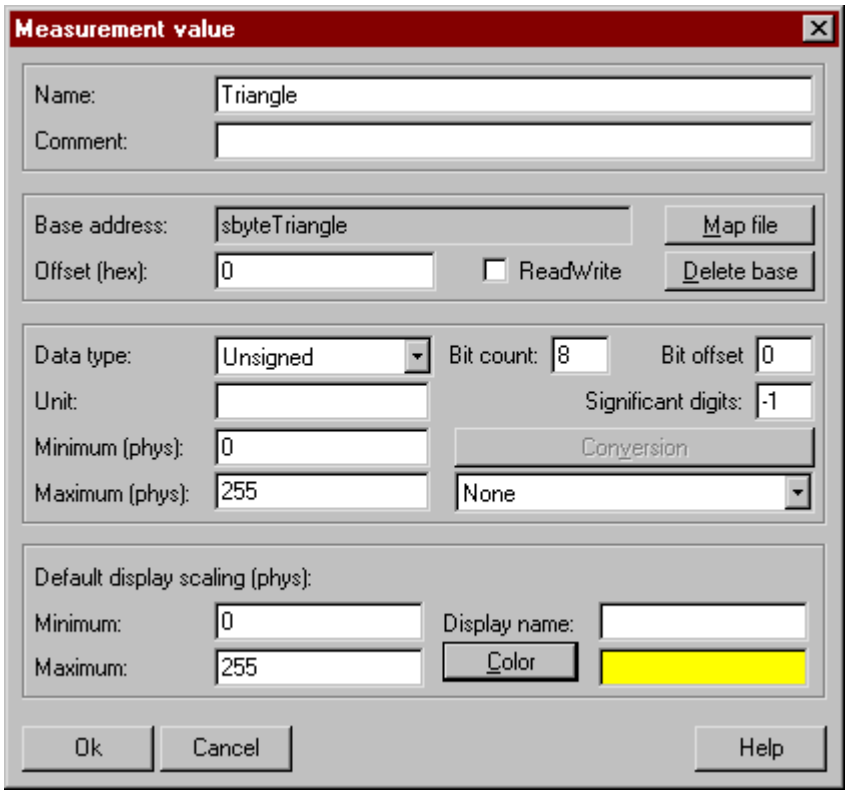


- ✓ click **<Close>**

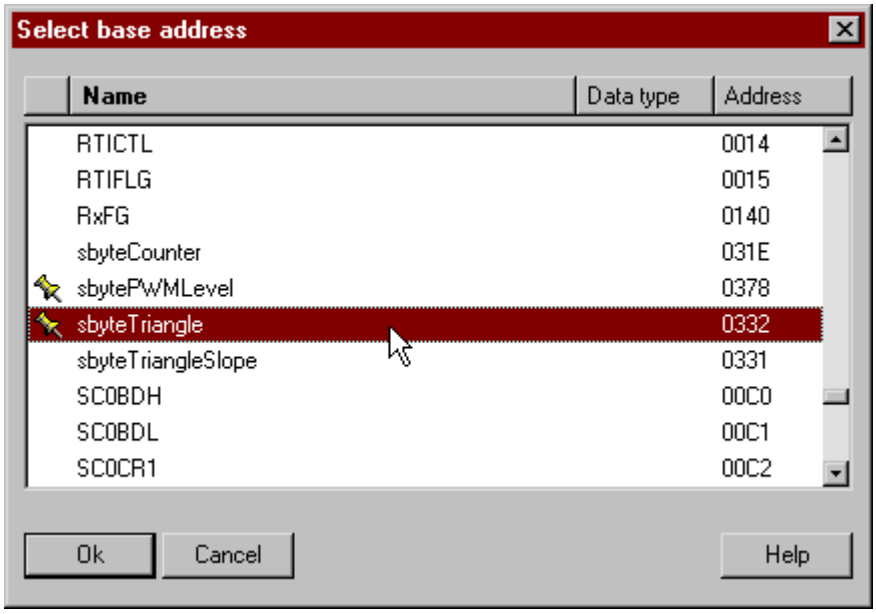
2.3 Adding a Measurement Signal to the Controller Database

CANape Graph has created an empty controller database. In this section we will add a measurement signal into the controller database. A measurement signal can be e.g. an output variable from the ECU device. In our example the **"Triangle"** signal will be added. 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 Graph User Manual.

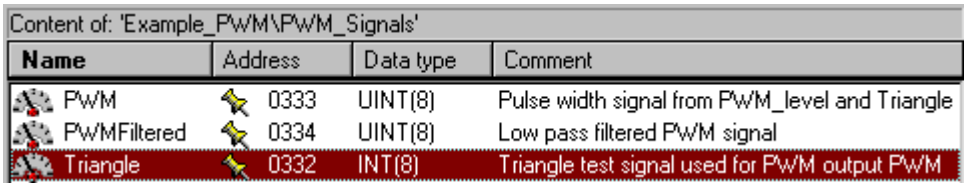
- ✓ To open the controller database with the database editor, the  symbol in the toolbar must be clicked
- ✓ Right-click in the right empty space and select **"New | Measurement value"**



- ✓ a name for the measurement object (here: Triangle) must be entered. A comment can also be entered, e.g. "Triangle test signal used for PWM output"
- ✓ to assign a linker map object to the measurement object "Triangle", the button **<Mapfile>** has to be clicked and the correct linker map object must be selected:



- ✓ select the linker map object: sbyteTriangle
- ✓ click **<Ok>**. On the right side the following can be seen:



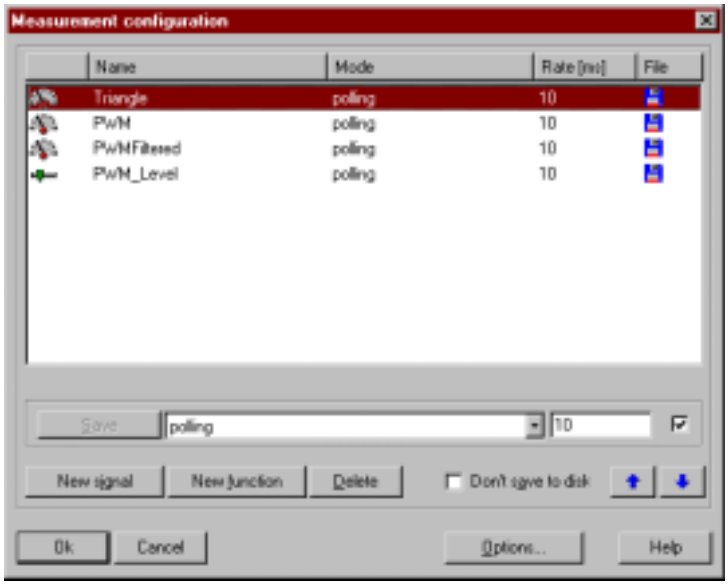
- ✓ exit the database editor with the  symbol

2.4 Measuring Signals in Polling Mode

In this chapter the **"Triangle"** signal will be added in the CANape Graph measurement list and this data will be acquired in the polling mode. Next a graphic window to display the triangle signal during the running measurement will be configured.

- ✓ the measurement list must be opened with the  symbol in the toolbar
- ✓ click **<New signal>** and select in the controller database the "Triangle" measurement signal.
  Press **<Apply>**

close the controller database window. The **“Triangle”** measurement signal will now appear in the Measurement configuration list, be sure that the data acquisition mode is set to **“polling”** and enter a rate, e.g. 10ms



- ✓ click **<Ok>**
- ✓ a new graphic window in CANape Graph can be created by right-clicking and selecting the entry **“Graphic window”**
- ✓ the signal **"Triangle"** can be added by right-clicking the empty graphic window and selecting the entry **“Insert measurement signal”**
- ✓ the measurement can be started by pressing the **F9** key or by clicking the  symbol in the toolbar
- ✓ a triangle signal should now be displayed in the graphic window:

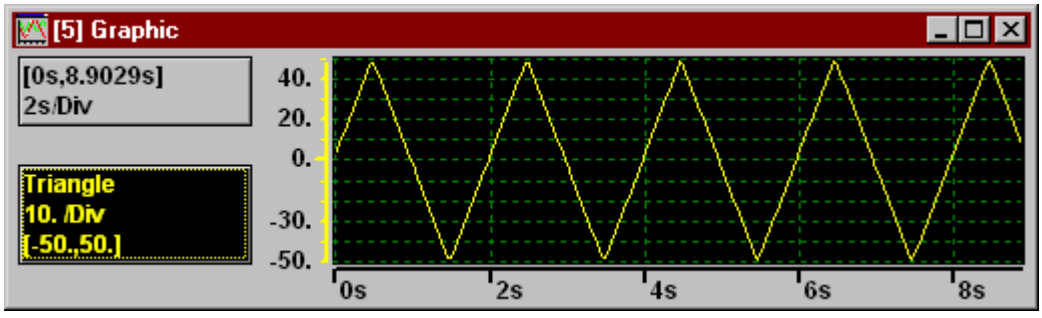


Figure 5: triangle signal displayed in graphic window

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

2.5 Enabling the Synchronous Data Acquisition Mode (DAQ)

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

The synchronous data acquisition mode has to be enabled in the *ccppar.h* file:

```
…
…
/*----------------------------------------------------------------------------*/
/* CCP Data Acquisition Parameters */

#define CCP_DAQ                /* Enable synchronous data acquisition in ccpDaq() */
#define CCP_MAX_ODT 3          /* Number of ODTs in each DAQ list */
#define CCP_MAX_DAQ 2          /* Number of DAQ lists */

/*----------------------------------------------------------------------------*/
/* CCP Options */

/* Use the transmit queue in CCP.C */
/* Complete sampling is done in ccpDaq(x) and the messages are
   written into the queue */
#define CCP_SEND_QUEUE



/* Indicate queue overruns in the msb of pid */
/* Will be displayed in CANape's status bar if CANAPE.INI: [asap1a] check_overflow=1 */
#define CCP_SEND_QUEUE_OVERRUN_INDICATION
…
…
```

The *ccpDaq(x)* function calls must be added in the ECU source code in all the places where the ECU has to transmit the measurement signals. In this CCP Test demo sample the *ccpDaq(x)* function will only be called in one place, in the interrupt service routine for Timer_0. The timer interrupt event has been programmed in this example to occur every 10 ms, which means that the timer interrupt service routine is also executed every 10 ms.

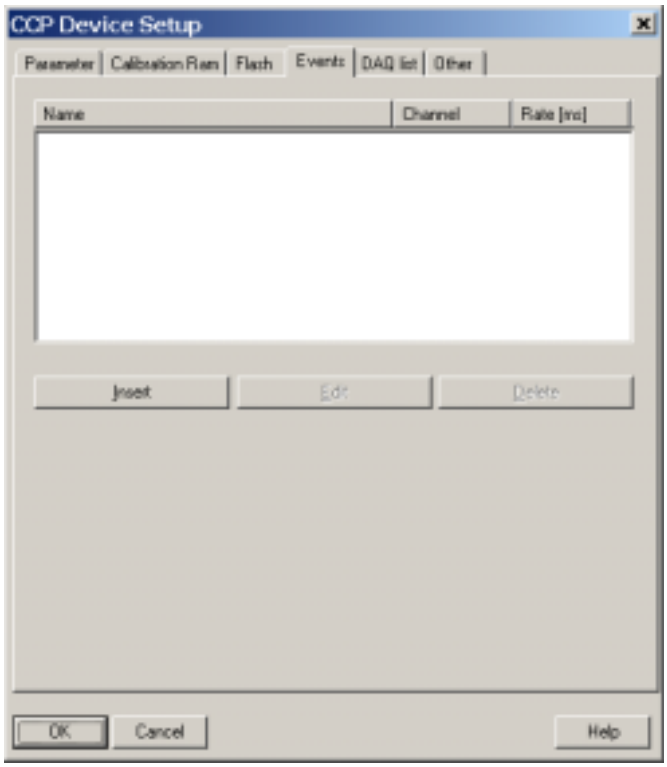
```
// ----------------------------------------------------------------------------
// Interrupt Service Routine for Timer_0 //
// ----------------------------------------------------------------------------
@interrupt void _timer_0(void)
{
 TC0 = TC0 + counter;           // incr. TimerCounter
 TFLG1 = 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 the whole project must be recompiled and the HC12 must be re-program with the new code.

2.5.1 Using the Synchronous Data Acquisition Mode in CANape Graph

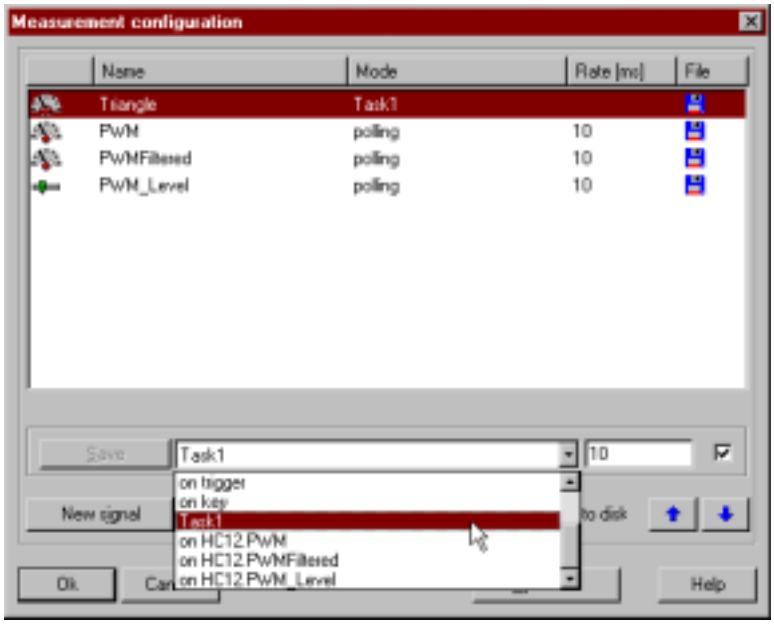
In CANape Graph all the used synchronous data acquisition event channels in the ECU must be defined. Then the measurement mode of the signals must be configured in the measurement list.

- ✓ start CANape Graph with a double click on the CANape icon on your desktop
- ✓ open the CCP Device Setup dialog with **"Tools |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)

- ✓ set **“Rate”** to 10ms (ccpDaq(1) is executed every 10ms in the timer interrupt service routine)
- ✓ click **<Ok>** to close the Events dialog
- ✓ click **<Ok>** 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 current value of the measurement object **“Triangle”** is transmitted via CAN and the new value is displayed in CANape Graph.



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 Graph 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 Graph 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 Graph 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 to CANape Graph.

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

Example for the 'cyclic' mode:

The default data transmission rate of Task1 is 10ms. The user selects 'cyclic' mode with a data transmission rate of 50 ms. At measurement start CANape Graph calculates the prescaler value of 5 (50/10) and configures a new DAQ list with that 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 Graph, decrement prescaler value to (4)

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

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

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

10ms passes: ccpDaq(1), is prescaler == 0 ?, YES!, transmit the new measurement values to CANape Graph, set the prescaler value = 5

If the user has configured some measurement signals in "Task1" mode, CANape Graph receives the new measurement values (e.g. Triangle) every 10ms. If some other measurement signals are configured in "cyclic" mode with a rate of 50 ms, CANape Graph receives the new measurement values of these signals every 50ms.

- ✓ 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

2.5.2 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 Graph (see Figure 6).

Example:

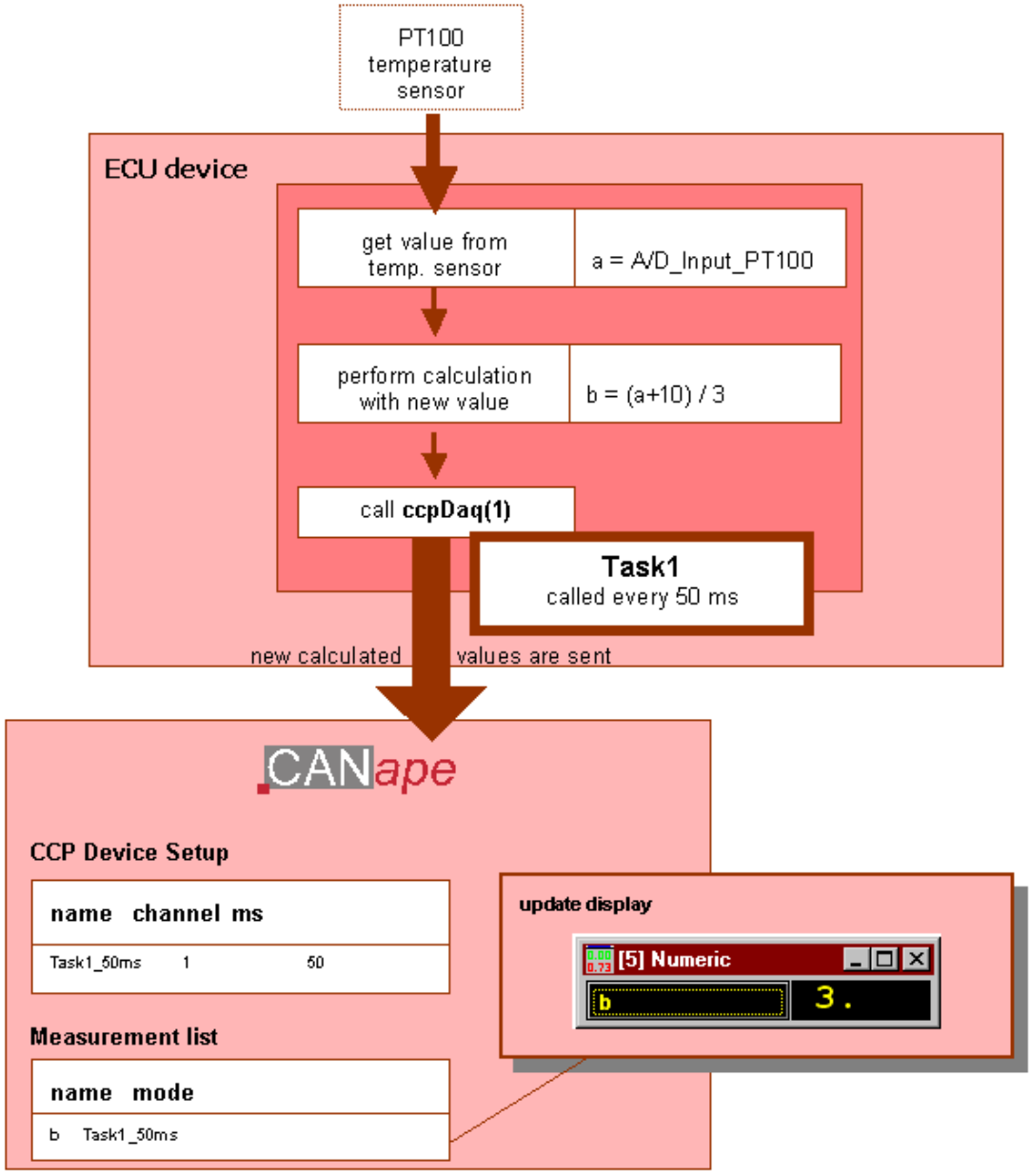
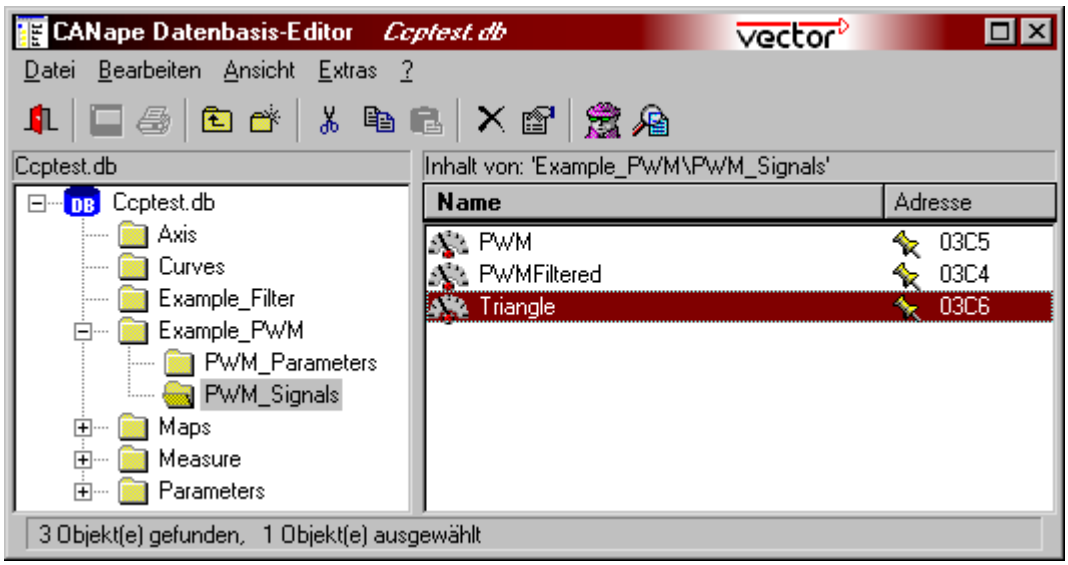
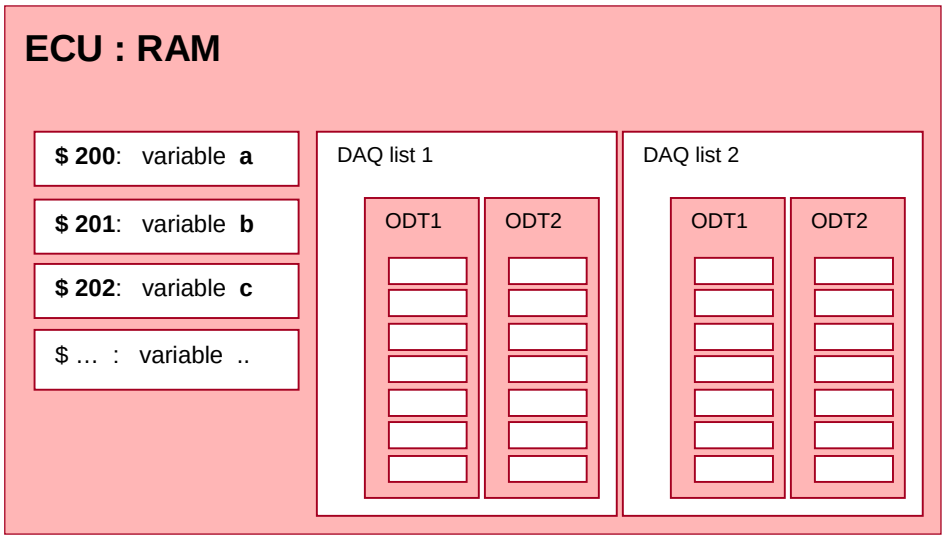


Figure 6: Measuring a signal from Task1 in synchronous data acquisition mode

The user defines the measurement objects from the ECU in the controller database and associates them with the correct address from the linker map file:

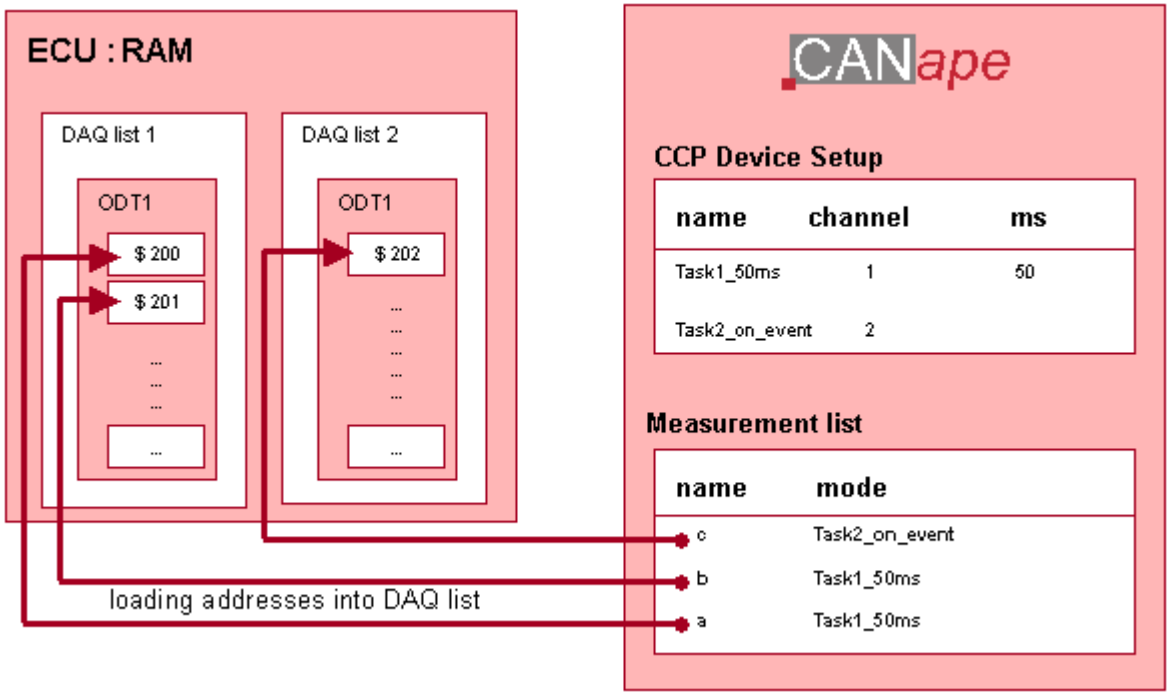


CANape Graph knows the address and data type of each measurement signal stored in the ECU via the controller database. The measurement signals are located in the RAM memory of the ECU. The CCP driver creates as many DAQ lists inside the ECU RAM as defined by CCP_MAX_DAQ in the *ccppar.h* file. A DAQ 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 Graph 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, CANape Graph starts the measurement.



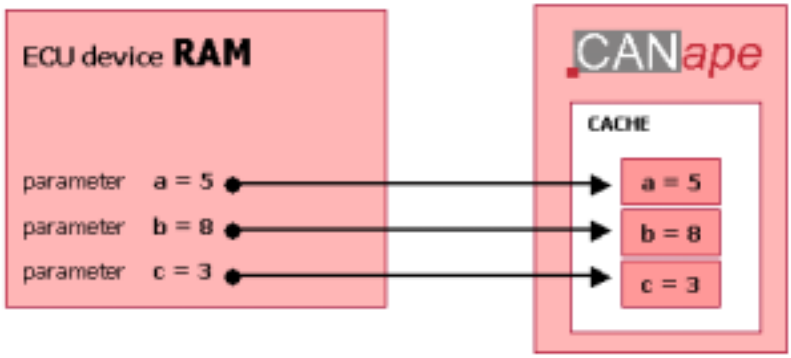
While the measurement is running, the application in the ECU calls **ccpDaq(x)**. The *ccpDaq(x)* generates a CAN message for each ODT of the DAQ list x, stores the created CAN messages in a send queue and starts to transmit the CAN DTO messages to CANape Graph.

2.6 Enabling the Checksum Calculation

In Section 1.7 the purpose of the checksum calculation has been explained. If the checksum calculation in the ECU is not implemented, then the parameters can only be changed by the user in the online mode. To be able to change parameter values also during the offline mode, the checksum calculation in the ECU and the cache memory (also called mirror memory) in CANape Graph have to be enabled.

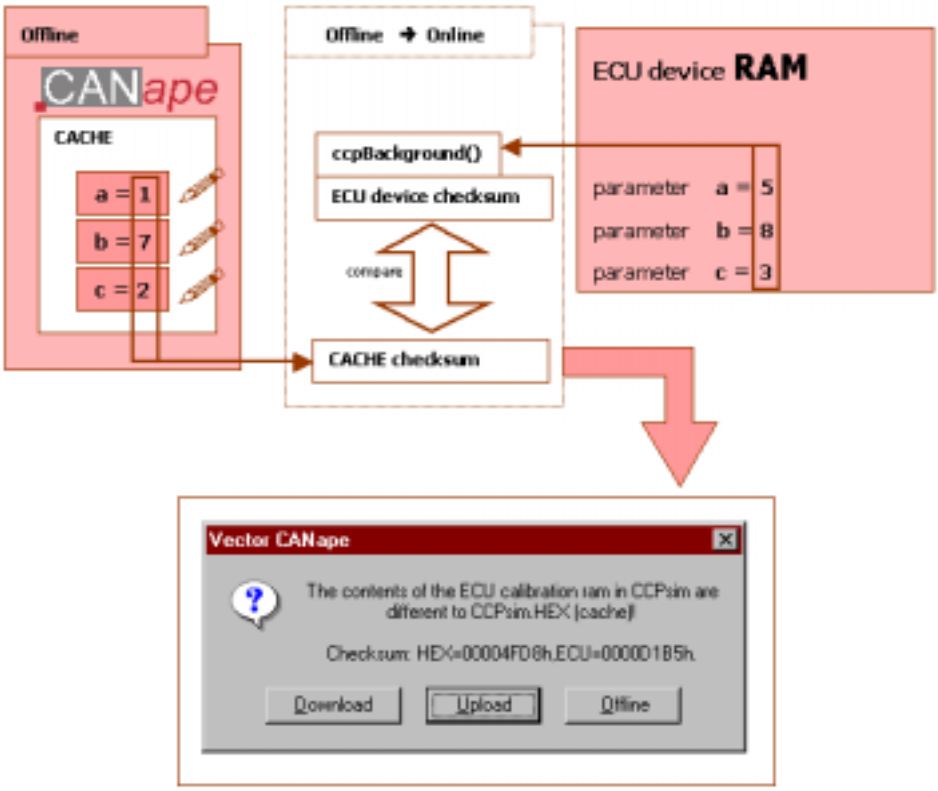
The prerequisite for the checksum calculation is that a few parameters are defined in the ECU. In our example we will use Curve_1 (shared X axis), Curve_2 (shared Y axis) and a Map_8x8 as parameters.

After enabling the cache memory, CANape Graph 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 Graph 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), then the values of the parameters can also be changed, but only the values of the cache memory are changed. When the user changes back to the online mode, CANape Graph compares the checksum of the cache memory with the checksum from the microcontroller. The checksum of the ECU is calculated by the **ccpBackground()** function. If the checksums are not equal, CANape Graph opens a message box.



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

2.6.1 Enabling the Checksum Calculation in the ECU

In this section the checksum calculation for the CCP Test demo sample will be enabled. In this example the checksum type WORD will be used. 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 enabled.

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

```
…
…
/* Implement the memory checksum feature */
/* The checksum will be calculated in ccpBackground() */
/* This may be implementation-specific */
#define CCP_CHECKSUM
#define CCP_CHECKSUM_TYPE WORD

/* Use a 16 bit CRC algorithm */
/* Note:
   This will need an additional 512 bytes of ROM
   CCP_CHECKSUM_TYPE has to be WORD !
*/
/* #define CCP_CHECKSUM_CCITT*/
…
…
```

Follow these steps:

- ✓ locate a place in the application which is called periodically (here: in the endless loop in ccp_test.c) and add the following line:

  `ccpBackground();`

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

2.6.2 Enabling the Checksum Calculation and the Cache in CANape Graph

- ✓ CANape Graph must be started via the icon on the desktop
- ✓ the CCP Device Setup dialog must be opened via **“Tools| Driver configuration...”** and select the tab **“CCP”**
- ✓ be sure the **“Enable”** checkbox is checked in the **“Checksum”** section, and set the **“Method”** to **ADD_12**
- ✓ change to the **“RAM”** tab

- ✓ in this dialog the calibration RAM areas of the ECU can be defined and the cache in CANape Graph 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 $ 300 (size = 1 x 8 bytes)
  curve_2_8_uc $ 308 (size = 1 x 8 bytes)
  map_1_8_8_uc $ 310 (size = 8 x 8 bytes)

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

  CCP TEST example: start address = 300h size = 50h

- ✓ click **<Ok>**

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

2.7 Enabling "Flash Kernel Download" in the ECU

- ✓ ***ccppar.h*** must be edited and the following define must be enabled:

```
/* Activate the flash programming kernel download */
#define CCP_BOOTLOADER_DOWNLOAD
```

- ✓ save ***ccppar.h***

2.7.1 Enabling "Flash Kernel Download" in CANape Graph

- ✓ start CANape Graph via the CANape CCP_Test icon

- ✓ click **"Tools | Driver configuration…"** in the main menu

- ✓ select the device, e.g. HC12_Vector and click **<Ok>**

- ✓ change to tab **"FLASH"** and insert flash sections:


CCP Device Setup
CCP
RAM
FLASH
Events
Other
Start
End
Size
Concat
01000h
07FFFh
07000h
0
08000h
0BFFFh
04000h
0
Insert
Modify
Delete
Flash signature
Enable flash signature
Address
hex
Length
Page switching
Active
Copy to RAM
Flash selector
hex
Ram selector
hex
Flash options 1
0xFF Optimization
Reconnect
Flash options 2
CALRAM Offset
0
hex
Flashkernel
Ok
Cancel
Help

The CARD12.D60 evaluation board contains two flash sectors

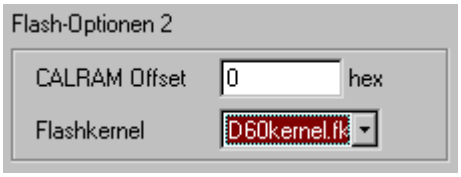
Flash Sector 1:	$ 1000 - $ 7FFF
Flash Sector 2:	$ 8000 - $ FFFF

The Flash Sector 2 contains the monitor program “TwinPEEK”, so the Flash Sector 2 space is reduced to $ 8000 - $ BFFF.

Insert two flash sections:

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

- ✓ check **“0xFF Optimization”** and **“Reconnect”**
- ✓ select the correct Flash Kernel for the used microcontroller, here: D60flashkernel.fkl



Note: All flash kernels are and have to be stored in the CANape\Exec folder

2.8 Enabling “Seed & Key” in the ECU

- ✓ edit ***ccppar.h*** and enable the following define:

```
/* Use GET_SEED and UNLOCK */
/* This is usually user dependant, just a skeleton here */
#define CCP_SEED_KEY
```

- ✓ save ***ccppar.h***

- ✓ insert the following two functions into the controller application (e.g. in ccp_can_interface.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

2.8.1 “Seed & Key” – DLL

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

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

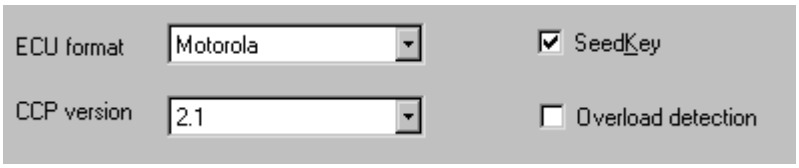
- ✓ Start the Microsoft® development environment,
- ✓ open project **"Seedkey1"**,
- ✓ edit the file *seedkey1.cpp* and implement the algorithm for calculating the "key" from the "seed":

```
  #include "stdafx.h"

  #define SEEDKEYAPI_IMPL
  #include "SeedKey1.h"

  #ifdef __cplusplus
  extern "C" {
  #endif

  unsigned long SEEDKEYAPI ASAP1A_CCP_ComputeKeyFromSeed(
    unsigned char *pSeed,
    unsigned short sizeSeed,
    unsigned char *pKey,
    unsigned short maxSizeKey,
    unsigned short *sizeKey)
  {
    /* Here: Simple example: Return the seed value as key */
    *((unsigned long*)pkey) = *((unsigned long*) pseed);
    *sizeKey = 4;

    return 1;

  }

  #ifdef __cplusplus
  }
  #endif  /* Implement here the algorithm for calculating the key from the seed
  */

  }
}
```

- ✓ Generate via the Microsoft® -Compiler the file *seedkey1.dll* and copy it into the project directory.

2.8.2 Enabling "Seed & Key" in CANape Graph

- ✓ start CANape Graph via the CANape CCP_Test icon
- ✓ click **"Tools | Driver configuration"**, select the ECU and check **"SeedKey"**



- ✓ Exit CANape Graph

If at the next CANape Graph program start, the SEED&KEY DLL will not be found or is the wrong one, then the user is not authorized for doing measurement, nor calibration, nor flash programming.

2.9 C Code Generation for Calibration Data Objects

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

2.9.1 Template Files

The code generation process is based on templates, which contains 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 will be 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 according 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.

2.9.2 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).

2.9.3 Template Example

Template curve.templ used in CANape Graph

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

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

//// End
```

Generated C Code file:

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

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

//// End
```

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

The starting point is the ecu.c file with an origin curve (named KL1 in CANape Graph):

```
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 correct C code for ‘KL2’
- ✓ open CANape Graph project CCPTEST2.CNA
- ✓ select curve **KL1** and change some parameters

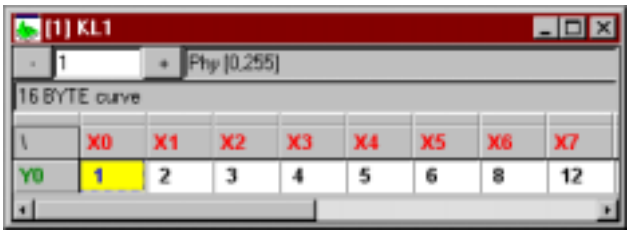


Figure 7: original values

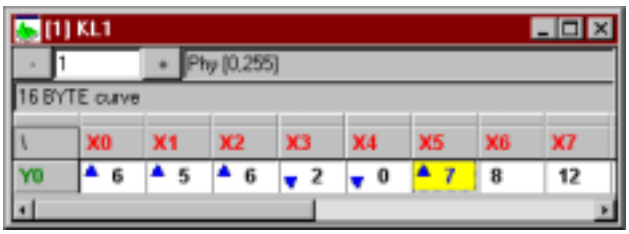
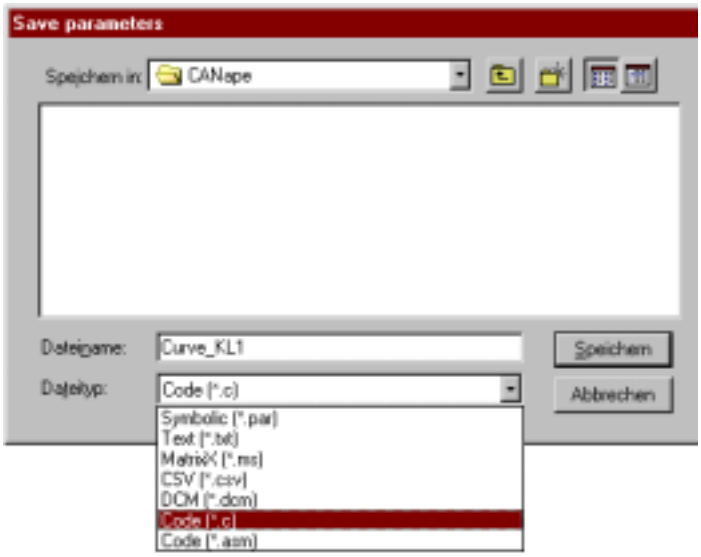


Figure 8: changed values

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



- ✓ click **<Save>** and press **<Ok>**
- ✓ The generated file looks like this:

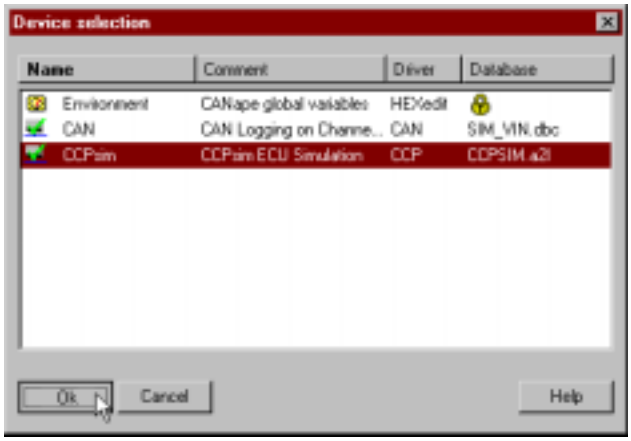
```
/* ------------------------------------------------------------------------*
///// Begin "curve.templ" for KL1
volatile unsigned char KL1[16] = /* 16 BYTE curve */
  { 6, 5, 6, 2, 0, 7, 8, 12, 10, 7, 2, 7, 10, 9, 8, 3};

//// 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};
…
…
```

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

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



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

3.0 CCP_TEST Demo Sample Files

SOURCE FILES

ccp_test.c
main application,
initializes LED on PORTH, Pin 7
initializes timer
initializes CAN driver
initializes Vector CCP Driver
checks if new CRO message is received, send DTO

ccp_can_interface.c
needed CCP functions,
irq_dummy interrupt (used by vectable.c)

boot_can.c
free CAN driver,
ccpBootInit(), initializes CAN driver
ccpBootTransmitCrm(), sends a CAN message
ccpBootReceiveCro, receives a CAN message
ccpBootTransmitCrmPossible, checks if ECU is ready to send new message

ccp.c

Vector CCP Driver

ecu.c

ECU simulation with some parameters etc.

timer.c

Timer_0. Calls ccpDaq(1) frequently by timer interrupt

vectable.c

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

4.0 CARD12.D60 Evaluation Board

The CARD12.D60 evaluation board used in this application note can be obtained at:

www.elektronikladen.de/en_card12.html

5.0 Additional Resources

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 Graph Manual V4.0**
   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)

6.0 Contacts

Vector Informatik GmbH
Ingersheimer Straße 24
70499 Stuttgart
Germany
Tel.: +49 711-80670-0
Fax: +49 711-80670-111
Email: info@vector-informatik.de

Vector CANtech, Inc.
39500 Orchard Hill Pl., Ste 550
Novi, MI 48375
Tel: (248) 449-9290
Fax: (248) 449-9704
Email: info@vector-cantech.com

VecScan AB
Fabriksgatan 7
412 50 Göteborg
Sweden
Tel: +46 (0)31 83 40 80
Fax: +46 (0)31 83 40 99
Email: info@vecscan.com

Vector France SAS
168 Boulevard Camélinat
92240 Malakoff
France
Tel: +33 (0)1 42 31 40 00
Fax: +33 (0)1 42 31 40 09
Email: information@vector-france.fr

Vector Japan Co. Ltd.
Nishikawa Bld. 2F, 3-3-9 Nihonbashi
Chuo-ku Tokyo 103-0027
Japan
Tel: +81(0)3-3516-7850
Fax: +81-(0)3-3516-7855
Email: info@vector-japan.co.jp