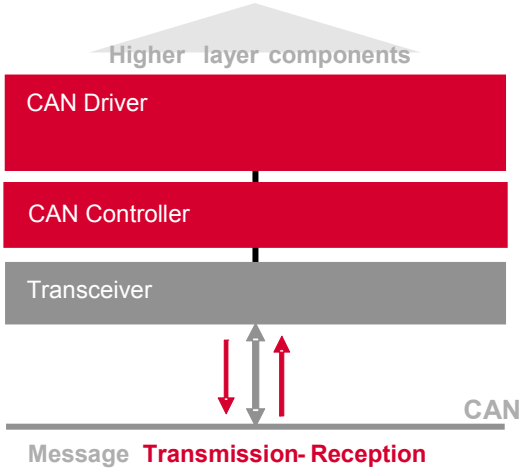
Vector CAN Driver

User Manual

(Your First Steps)

Version 2.4

Vector Informatik GmbH, Ingersheimer Str. 24, 70499 Stuttgart
Tel. 0711/80670-0, Fax 0711/80670-399, Email can@vector-informatik.de
Internet http:\\www.vector-informatik.de



Authors:	Klaus Emmert
Version:	2.4
Status:	released (in preparation/completed/inspected/released)

History

Author	Date	Version	Remarks
Klaus Emmert	2001-05-11	0.1	First version of the User Manual
Klaus Emmert	2001-08-11	0.4	Technical and linguistic revision
Klaus Emmert	2001-09.21	0.6	Revision (pretransmit)
Klaus Emmert	2001-10-10	0.6a	Error in description of a message, how to enter the manufacturer type to the example data base.
Klaus Emmert	2001-12-14	1.0	Linguistic revision
Klaus Emmert	2002-09-25	1.4	Linguistic corrections and little adaptations
Klaus Emmert	2003-07-16	1.5	Warning added for example code usage
Klaus Emmert	2004-10-26	1.6	New Layout, example dbc file deleted and description modified, description for CANgen and the new Generation Tool GENy, new Symbols
Klaus Emmert	2006-05-30	1.7	Updated dialog for bus timing register setup
Klaus Emmert	2006-09-08	1.8	Page number of Index, headline numbering
Klaus Emmert	2007-02-20	1.9	Issues in Word Hyperlinks
Klaus Emmert	2007-07-26	2.0	Issues in steps introduction, some typos.
Klaus Emmert	2007-09-06	2.1	Typos and reference in TOC
Klaus Emmert	2007-10-29	2.2	Baudrate setting description
Klaus Emmert	2008-09-04	2.3	Include file for CAN Driver and GENy
Klaus Emmert	2009-08-26	2.4	Fix: v_inc.h must not be changed manually.

Motivation For This Work

The CAN Driver is the only component among the CANbedded Software Components that is directly connected with the CAN Controller hardware. It is the foundation for all other CANbedded Software Components.

The first target for you is to get the CAN Driver running, to see receive and transmit messages on the bus.

WARNING

All application code in any of the Vector User Manuals is for training purposes only. They are slightly tested and designed to understand the basic idea of using a certain component or a set of components.

Contents

Illustrations

1 Welcome to the CAN Driver User Manual

1.1 Beginners with the CAN Driver start here ?

You need some information about this document? → see Chapter 2

Getting started → see Chapter 3

9 Steps for the CAN Driver → see Chapter 6

1.2 For Advanced Users

Start reading here. → see Chapter 5

1.3 Special topics

Strategies for receiving a CAN message → see Chapter 6.9.1

Strategies for sending a CAN message → see Chapter 6.9.2

An exercise → see Chapter 7.1

1.4 Documents this one refers to…

TechnicalReference_CANDriver.pdf

TechnicalReference_CAN_xxx.pdf

2 About This Document

This document gives you an understanding of the CAN Driver. You will receive general information, a step-by-step tutorial on how to include and use the functionalities of the CAN Driver.

For more detailed information about the CAN Driver and its API refer to the Technical Reference (TechnicalReference_CANDriver.pdf) and the hardware specific references TechnicalReference_CAN_xxx.pdf (e.g. TechnicalReference_CAN_HC12.pdf).

2.1 How This Documentation Is Set-Up

Chapter	Content
Chapter 1	The welcome page is to navigate in the document. The main parts of the document can be accessed from here via hyperlinks.
Chapter 2	It contains some formal information about this document, an explanation of legends and symbols.
Chapter 3	An introduction to the files, the tools and information necessary to understand the descriptions in the following chapters.
Chapter 4	Here you find some more insight in the CAN Driver about receiving and transmitting messages, the CAN controllers and the data structure.
Chapter 5	A step-by-step guide to establish CAN communication on an ECU for the first time. Follow the 9 steps to get the answer to most of your questions and problems..
Chapter 6	Here you find a problem to solve to check your understanding of the CAN Driver and its functions.
Chapter 7	In this last chapter there is a list of experiences with the CAN Driver.

2.2 Legend and Explanation of Symbols

You find these symbols at the right side of the document. They indicate special areas in the text. Here is a list of their meaning.

Symbol	Meaning
	The building bricks mark examples.
	You will find key words and information in short sentences in the margin. This will greatly simplify your search for topics.
	The footprints will lead you through the steps until you can use the described Vector CAN Driver.
	There is something you should take care about.
	Useful and additional information is displayed in areas with this symbol.
	This file you are allowed to edit on demand.
	This file you must not edit at all.
	This indicates an area dealing with frequently asked questions (FAQ).

These areas to the right of the text contain brief items of information that will facilitate your search for specific topics.

3 ECUs and Vector CANbedded Components – An Overall View

3.1 Network Data Base File (DBC)

Normally the different ECUs in a modern vehicle are developed by different suppliers (SUPPLIER X). All ECUs within the same bus system (❶ or ❷) use the same data base (dbc file) to guarantee that the ECUs will work together later on in the vehicle.

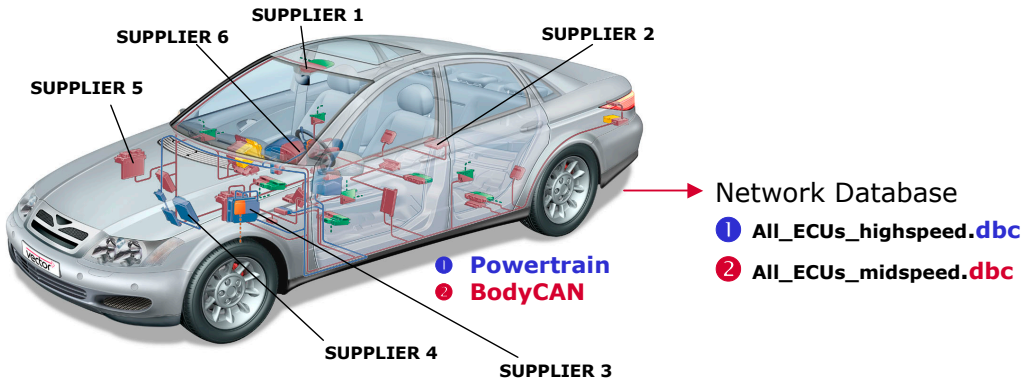


Figure 3-1 A Modern Vehicle With Body CAN And PowerTrain Bus

The dbc file is designed by the vehicle manufacturer and distributed to all suppliers that develop an ECU. Thus every supplier uses the SAME dbc file for one vehicle platform and one bus system (powertrain, body CAN etc.) to guarantee a common basis for development.

There is the same dbc file per bus system (high speed, low speed, etc) for all suppliers to guarantee a common basis for development.

The dbc file contains e.g. information about every node in the network, the messages/signals each node has to send or to receive. The distribution of the signals among the messages is stored in the DBC file, too.

For example: every ECU has to know that a 1 in bit 7 in the 4th byte of the message 0x305 means “Ignition Key” on/off.

4 CANbedded Software Components

The vector CANbedded environment consists of a number of adaptive source code components that cover the basic communication and diagnostics requirements in automotive applications.

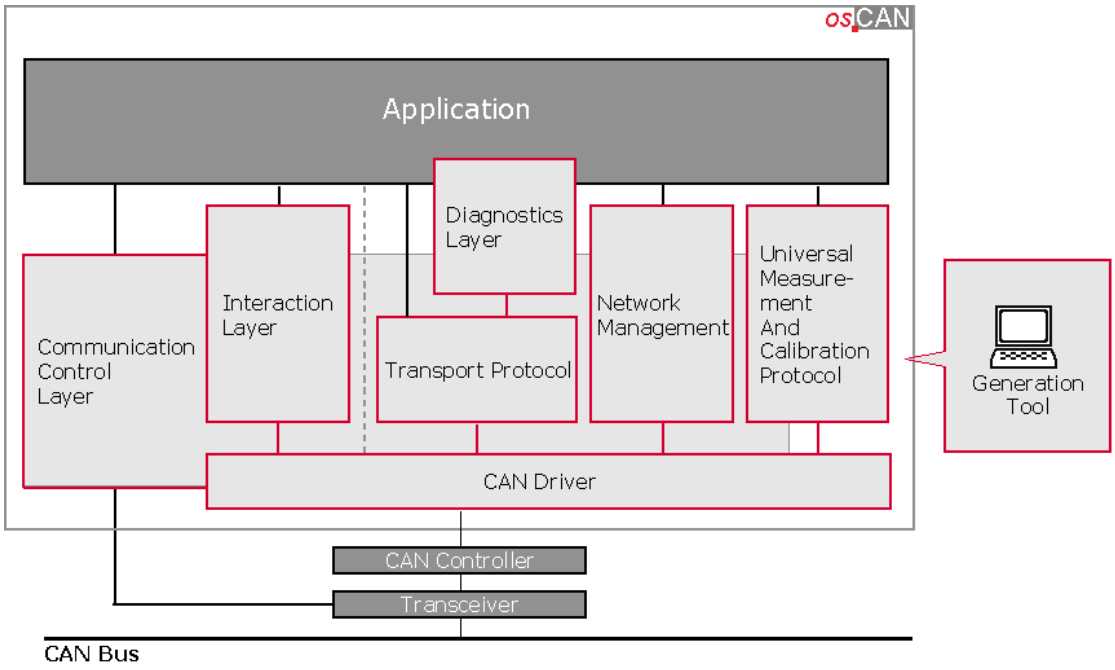


Figure 4-1 CANbedded Software Components

CAN Driver
The CAN Driver handles the hardware specific CAN chip characters and provides a standardized application interface.

Interaction Layer
The Interaction Layer (IL for short) is responsible for the transmission of messages according to specified rules, monitoring receive messages, timeout monitoring, etc. It provides a signal oriented application interface for the application.

Transport Protocol
The CAN protocol is restricted to 8 data bytes per message. But in some cases (e.g. diagnostics) you need to exchange much more than 8 data bytes. The segmentation of the data, the monitoring of the messages and the timeouts is done by the Transport Protocol (TP for short).

Diagnostics
Diagnostics Layer according to ISO14229 / ISO14230 (Keyword Protocol 2000).

Network Management
The Network Management (NM for short) is the component to control the bus, to synchronize the transition to bus sleep, error recovery after bus-off, etc.

Communication Control Layer (CCL)
The CCL provides an integration environment for the CANbedded Software Components, an abstraction for different vehicle manufacturers, microcontrollers, CAN Controllers and compilers/linkers. It also provides a global debug mechanism.

Universal Measurement and Calibration Protocol (XCP)
This is the Software Component for measurement and calibration on several bus systems. To mention some feature: read and write access to various memory locations or flash programming.

Generation Tool
This is a PC tool for configuring the above listed components. The Generation Tool is driven by a network database file, DBC file for short.

The CANbedded Software Components are configurable and can be adapted to your specific needs via the Generation Tool.

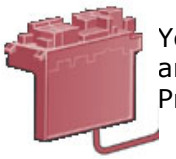
4.1 Generation Tool

The Generation Tool displays the complete set of ECUs in the network. In general you pick out the one you develop the software for. In special cases when you develop ECUs that are almost identical (e.g. door ECUs) you select more than one (so-called multiple ECUs).

For **your** ECU there are special requirements concerning the hardware and the functionality. I.e. the driver must be suitable for your hardware and the standard components must be adaptable for your project-specific needs. The means to do this is the Generation Tool.

The Generation Tool is a PC-Tool. It reads in the Network Data Base file (DBC) and offers you to select the node you are going to develop and has therefore all information about your ECU, the receive and transmit messages, the signals etc.

The Generation Tool generates files that contain this information (DBC, hardware and specific settings) and so complete the components core files and have to be included in the compile and link process.



Your Hardware Platform-
and Compiler information
Project specific component settings → Application Specific Data

Include the generated files in your system as shown.

You must not change the generated files by hand – the next generation process will delete these changes.

Right now there are two Generation Tools available, CANgen and the new Generation Tool called GENy. Which tool you have to use depends on you delivery and the project.

4.2 The Vector CAN Driver

A driver is a program to control a piece of hardware. In this case the Vector CAN Driver controls the CAN Controller and its registers.

4.2.1 Tasks of The Vector CAN Driver

The CAN Driver basically handles the reception and transmission of information via the CAN Bus and recognizes bus failure (bus off). The CAN Driver provides a standard application interface for the application.

Your application only has to use a set of predefined functions to control the CAN Driver and will be notified via interrupt about incoming information (messages). To control the incoming and outgoing data and to be notified of important events you have to add some service- and call-back-functions to your application (see 6.5).

For more detailed information about the CAN Driver please refer to the Vector CAN Driver Technical Reference (TechnicalReference_CANDriver.pdf and TechnicalReferfence_CAN_<hardwareplatform>.pdf, e.g. TechnicalReference_CAN_HC12.pdf).

4.2.2 Vector CAN Driver Files

The Vector CAN Driver consists of 3 sets of files, component files, generated files and configurable files

4.2.2.1 Component Files

Independent of the used Generation Tool
can_drv.c - can_def.h - v_def.h

You must not change these files at all.

4.2.2.2 Generated Files

Independent of the used Generation Tool

can_cfg.h - v_cfg.h

Only for CANgen

YourECU.c - YourECU.h

Only for GENy
can_par.c - can_par.h - drv_par.c - drv_par.h - v_par.h - v_par.c - v_inc.h

Do not change these files. You will lose the changes after the next generation process.

4.2.2.3 Configurable files

Independent of the used Generation Tool

can_inc.h

INC stands for include. Here you can add includes you need.. You have to include can_inc.h in every application C file where you need CAN functionality, followed by the include of YourECU.h.

The Generation Tool CANgen generates the signal and message access macros as well as the indication or confirmation flags to the file **YourECU.h**. GENy generates this to the file **can_par.h**.

4.2.3 Include The CAN Driver Into Your Application

Use the illustration in Figure 4-2 to include the files for the CAN Driver into your application correctly. Please keep to the including order to avoid errors while compiling or linking.

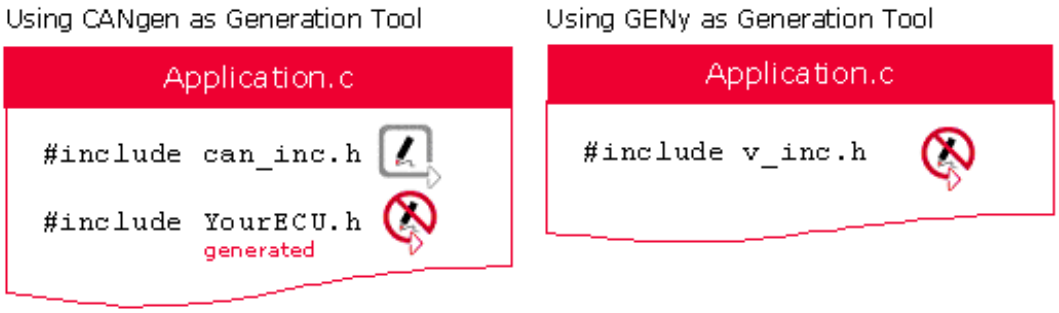


Figure 4-2 Order For Including Files

5 Vector CAN Driver– A More Detailed View

You access the signals relevant for your ECU with the macros generated by the Generation Tool.

The generated file sig_test.c contains a list of all access macros to the signals.

5.1 Information Package on the CAN Bus

As mentioned before, the information is exchanged between ECUs via the CAN bus. The maximum amount of data which can be exchanged is 8 data bytes and they are transmitted via a so-called **message**.

A message contains the ID (the “name” or number of the message), the DLC (data length code, i.e., the number of data bytes) and the data bytes .

A message on the CAN Bus can contain from 0 to 8 data bytes.

Every message is divided up into **signals**. A signal consists of 1 up to 64 bits. A signal cannot exceed the **message** boundary.

We do not consider signals here which are greater than 64 bits, as this involves the Transport Protocol.

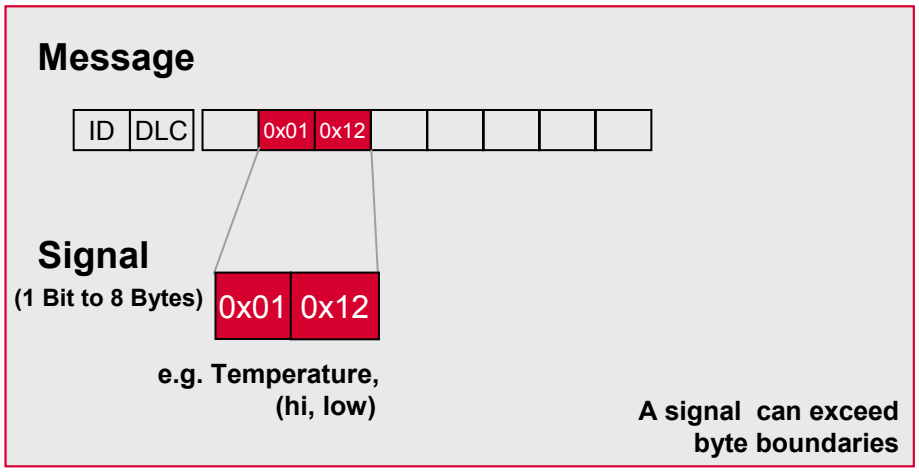


Figure 5-1 Message and Signal

Signals are assigned to messages by the vehicle manufacturer database engineer. This assignment is stored in the database (dbc file).

Normally you must not change the database (dbc file).

5.2 Storing Information Packages

The more you understand about data handling in the CAN Driver the better you are able to design your application. You have to know where the data is stored at a specific point in time to be able to access this data correctly.

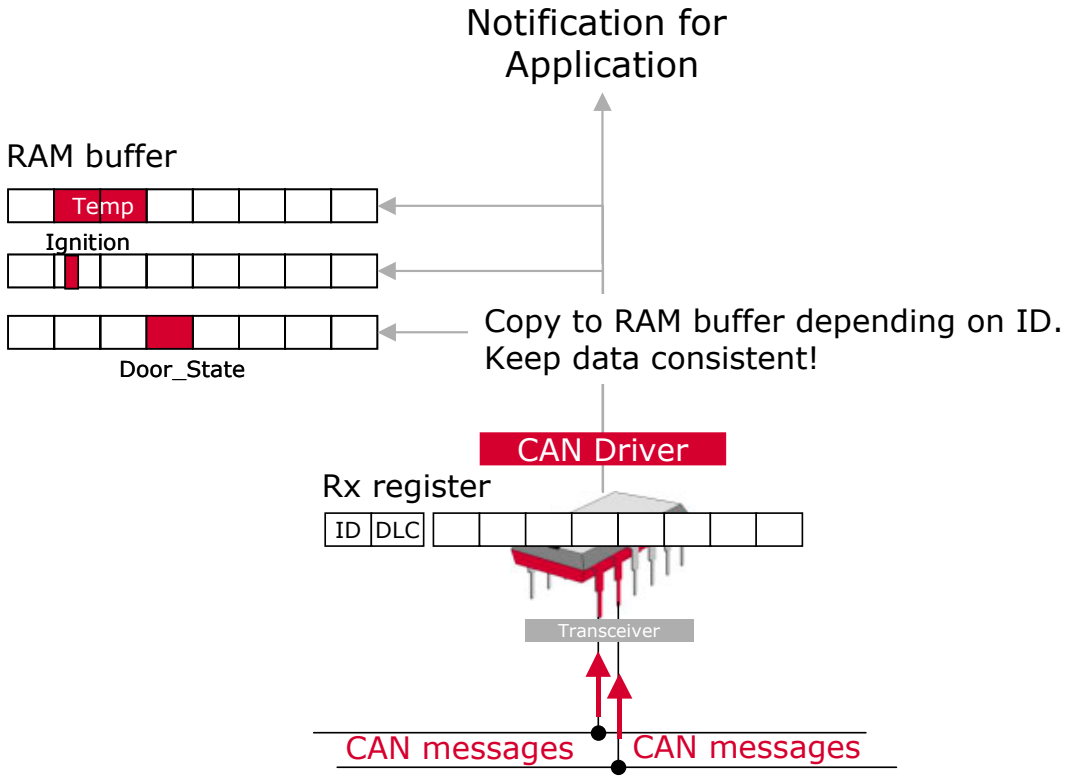


Figure 5-2 Rx Register, Data Structure and Notification

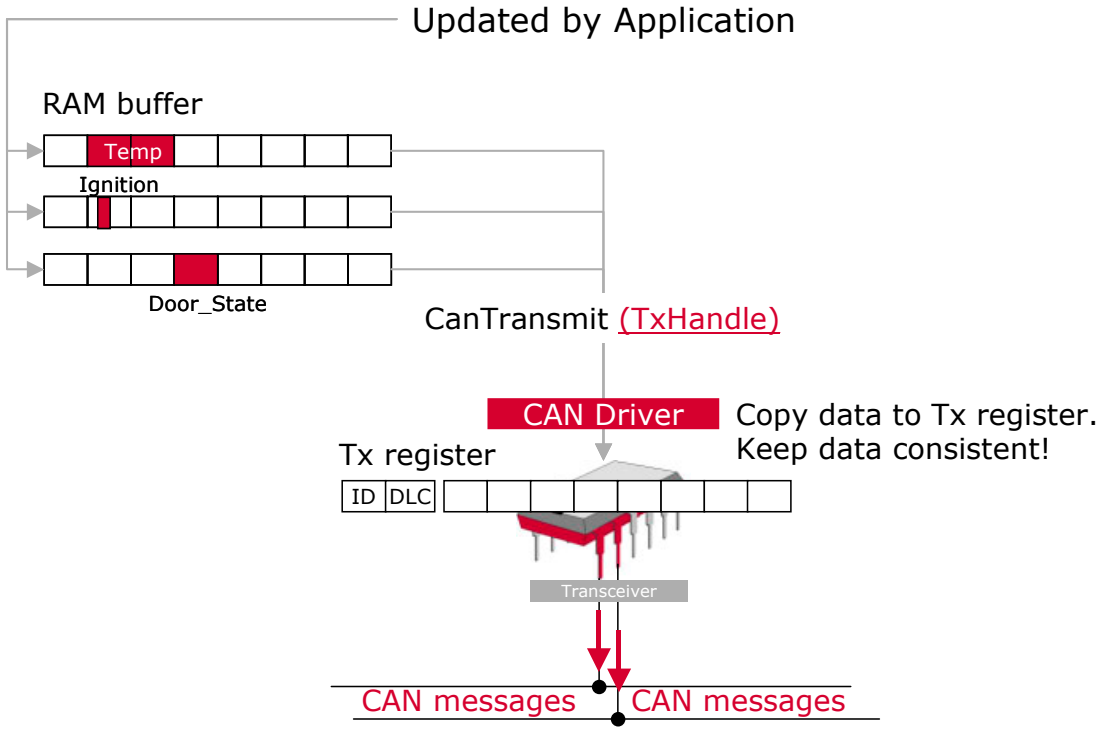


Figure 5-3 Tx Register, Data Structure and CanTransmit

There are 3 elements involved in storing information packages:

5.2.1 The Registers of the CAN Controller

Your CAN Controller has a receive register (Rx Register) and a transmit register (Tx Register).

Data is always received in the Rx Register of the controller. The data is written to the Tx Register immediately before a transmission.

You can access these two registers via the signal access macros containing the _CAN_ in the name (see YourECU.h if you use CANgen and can_par.h if you use GENy).

Later on you will see, that this special access to the hardware is possible only in the functions ApplCanMsgReceived and in the Precopy Functions for reception and in the Pretransmit Function for transmission.

The **signal** access to the Tx Register is dependent on the hardware, not all drivers support this feature.

5.2.2 The Data Structure Generated by the Generation Tool for Storing Message Data.

The Generation Tool defines variables to allocate memory for the data of the receive messages and the transmit messages (The data allocation is optional and can be switched off). In the receive procedure, the data will be copied from the Rx Register to the message-specific memory area (RAM). In the transmit procedure, you have to enter the current values in the variables and the driver will copy the data to the Tx Register as shown in Figure 5-2 and Figure 5-3.

The decision, whether to copy the data to or from the registers (receive and transmit) with **your own** function or whether you let the driver do the copying action is up to you and is decided by the return value of specific functions (precopy function, pre-transmit function, see 6.9).

The Generation Tool optimizes the memory consumption. The highest byte containing relevant signals determines the amount of bytes to be reserved for this message. In the picture below, the red areas and lines in the bytes show where relevant signal information is stored within the message.

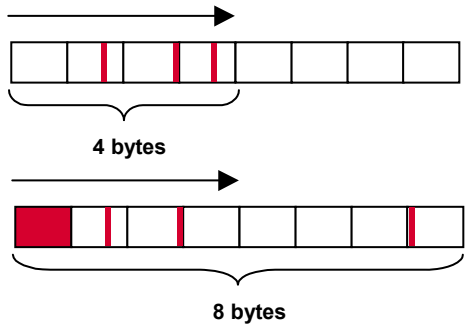


Figure 5-4 Memory optimization by the Generation Tool

The Generation Tool minimizes the amount of memory reserved for a message.

In the first message the 3rd byte contains the last signal (counting started with 0). Byte 4, 5, 6 and 7 have no relevant information for your ECU. In this case the Generation Tool only reserves 4 (0-3) bytes for this message.

The second message has information in the 7th byte, so the number of byte to be reserved is 8 (0-7).

5.2.3 Memory the Application Reserved for Signals.

Sometimes you only need one byte or even only one bit signal out of a message. To save RAM memory do not use a generated data structure. Use the precopy function (see 6.9.1.5) to copy the byte or the bit of the received message to the byte or the bit field you reserved in your application to hold this specific information.

6 CAN Driver in 9 Steps

STEP 1 : UNPACK THE DELIVERY
Follow the install shield, unpack the delivery and install the generation tool.

STEP 2: GENERATION TOOL AND DBC FILE
Configure the CAN Driver using the generation tool and an appropriate data base file (DBC)..

STEP 3: GENERATE FILES
The generation tool generates all necessary files for the CAN Driver.

STEP 4: ADD FILES TO YOUR APPLICATION PROJECT
To use the CAN Driver you have to add the CAN Driver files to your application project.

STEP 5: ADAPTATIONS FOR YOUR APPLICATION
Also adapt your application files to be able to use the functionality of the CAN Driver.

STEP 6: COMPILE AND LINK
Compile and Link your application project including the CAN Driver.

STEP 7: RECEPTION OF A CAN MESSAGE
Test the receiving path of the CAN Driver by sending a message to your ECU.

STEP 8: TRANSMISSION OF A CAN MESSAGE
Transmit your first CAN message using the CAN Driver service functions.

STEP 9: FURTHER ACTIONS AND SETTINGS
Topics above the very easy beginning.

6.1 STEP 1 Unpack the Delivery

The delivery of CANbedded Software Components normally comprises a Generation Tool and the source code of the software components.

You only have to start the

…_Setup.exe

and to follow the install shield wizard.

> We recommend creating a shortcut to the Generation Tool.

The Generation Tool generates files for your application. It is the connection between your hardware and settings, the requirements of your vehicle manufacturer and the other ECUs, your ECU has to communicate with.

Back to 9 Steps overview

6.2 STEP 2 Generation Tool and dbc File

Use new Generation Tool GENy, look there >>

6.2.1 Using CANgen as Generation Tool

When you start the Generation Tool you will see a window like Figure 6-1. This starting window is the main window of the Generation Tool.

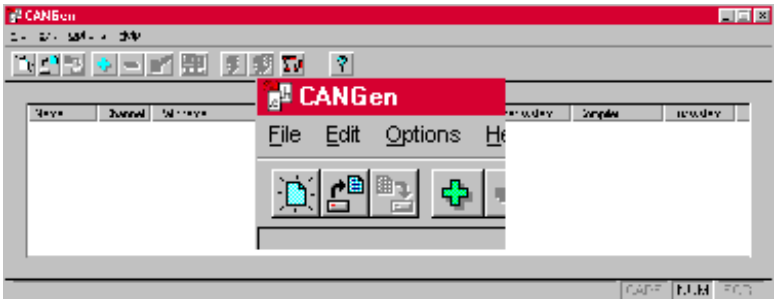


Figure 6-1 Add a dbc file

A click on the green plus (+) will open a new window to add your database (dbc file).

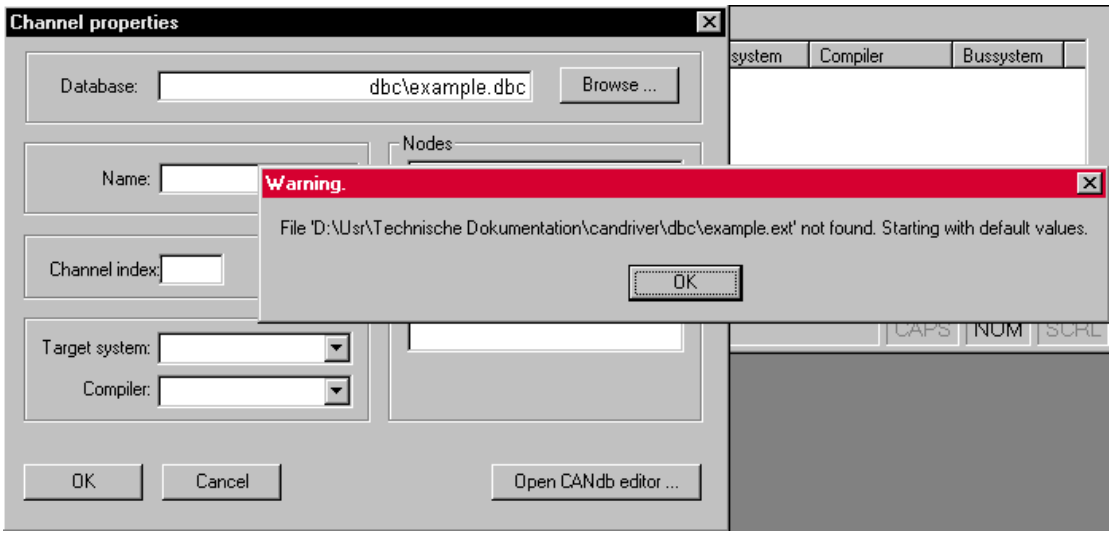


Figure 6-2 Warning – do not bother

Browse for your dbc file and select it. When you do this for the first time, the Warning above will occur. Ignore it, and just confirm with **OK**..

Select your node; choose your target system and compiler.

Since we are using a CAN Driver with just one channel, the channel index has to be left empty. When you have two or more channels, they will be distinguished via this index.

When you use the browse-button you will get an absolute path to the dbc file.

It is suggested that you use relative paths in order to be able to move your project more easily from one directory to another.

This warning occurs only if you open the dbc.file for the first time because of the extension file has still not been created. The extension file will contain your settings you do in the following.

The Name field is set to the name of the dbc file. You can change the name . It is only used in the list of your channels in figure 2-5.

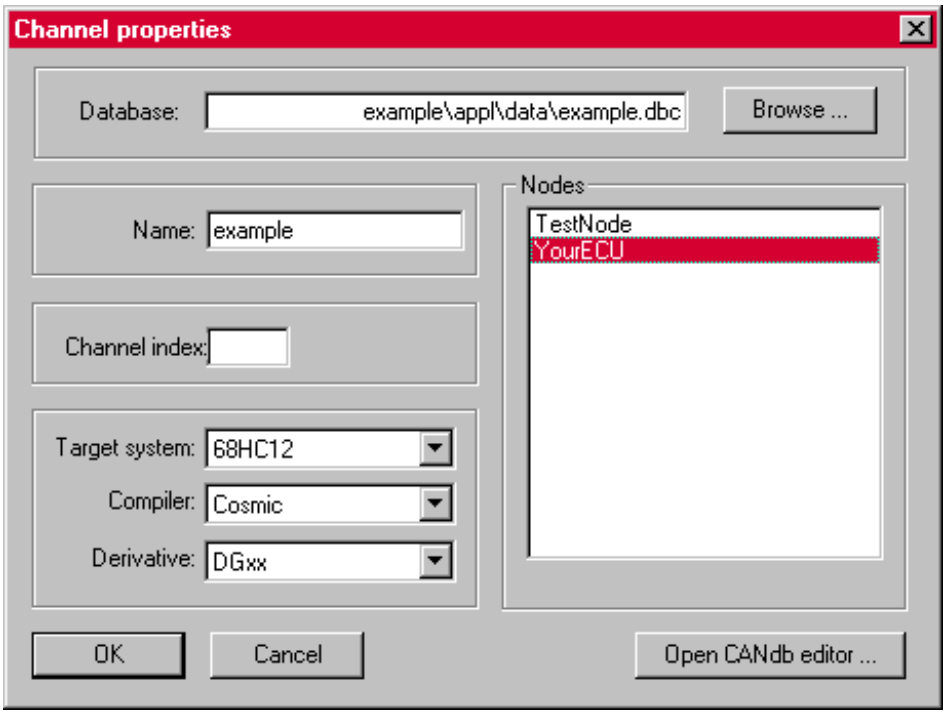


Figure 6-3 Channel properties

Confirm with OK and you will see your setting as text in the field of the starting window of the Generation Tool. When we save the configuration for the first time, we have to use the menu command **File/Save as**.

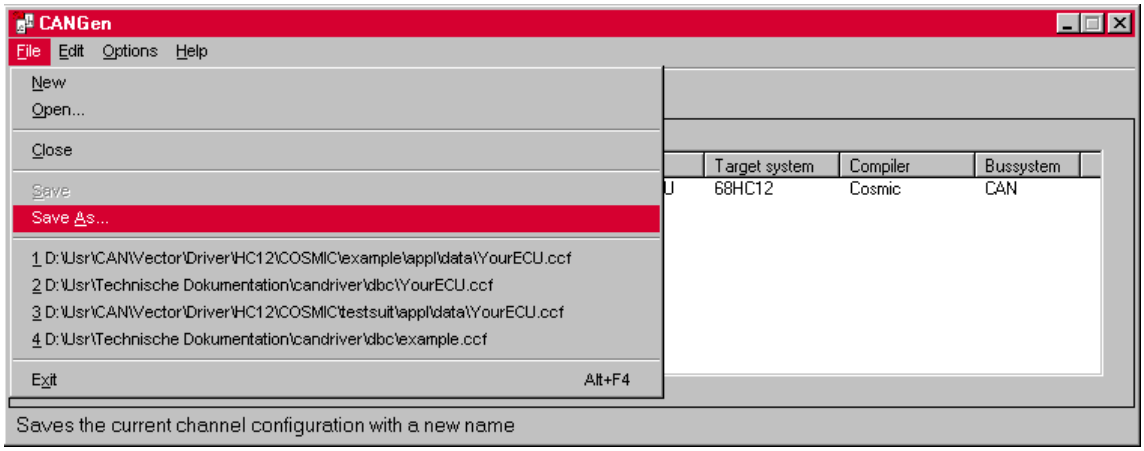


Figure 6-4 Save Setting For The First Time

Take a look at the directory where you stored the settings of the Generation Tool. There is your dbc file and new files (see note) only used by the Tool.

The following files belong to the Generation Tool:

<databasename>.dbc	(exampleDRV.dbc)
<databasename>.ext	(exampleDRV.ext)
<databasename_nodename>.msg	(exampleDRV_YourECU.msg)
<databasename_nodename>.sig	(exampleDRV_YourECU.sig)

(<databasename_nodename>.fms fms only when using a full CAN controller.	(exampleDRV_YourECU.fms))
yourECU.ccf	this is the project file for the Generation Tool.

Now you can open your settings in the normal way (not via the dbc file) by opening the file YourECU.ccf.

When your vehicle manufacturer changes the dbc file, just copy all files important for the Generation Tool into a new directory, delete the old dbc file and copy the new dbc file in this new directory. Rename the old ext file with the name of the new dbc file. This preserves your application-specific settings.

This does not work when the target system has changed!!!

The checkbox Generate only bit and byte signal macros is only used for very special applications. If you have any doubt, do not choose this.

Now we shall make additional settings for the component CAN Driver. Use this button[] to open the Generation Options and to enter the following dialog.

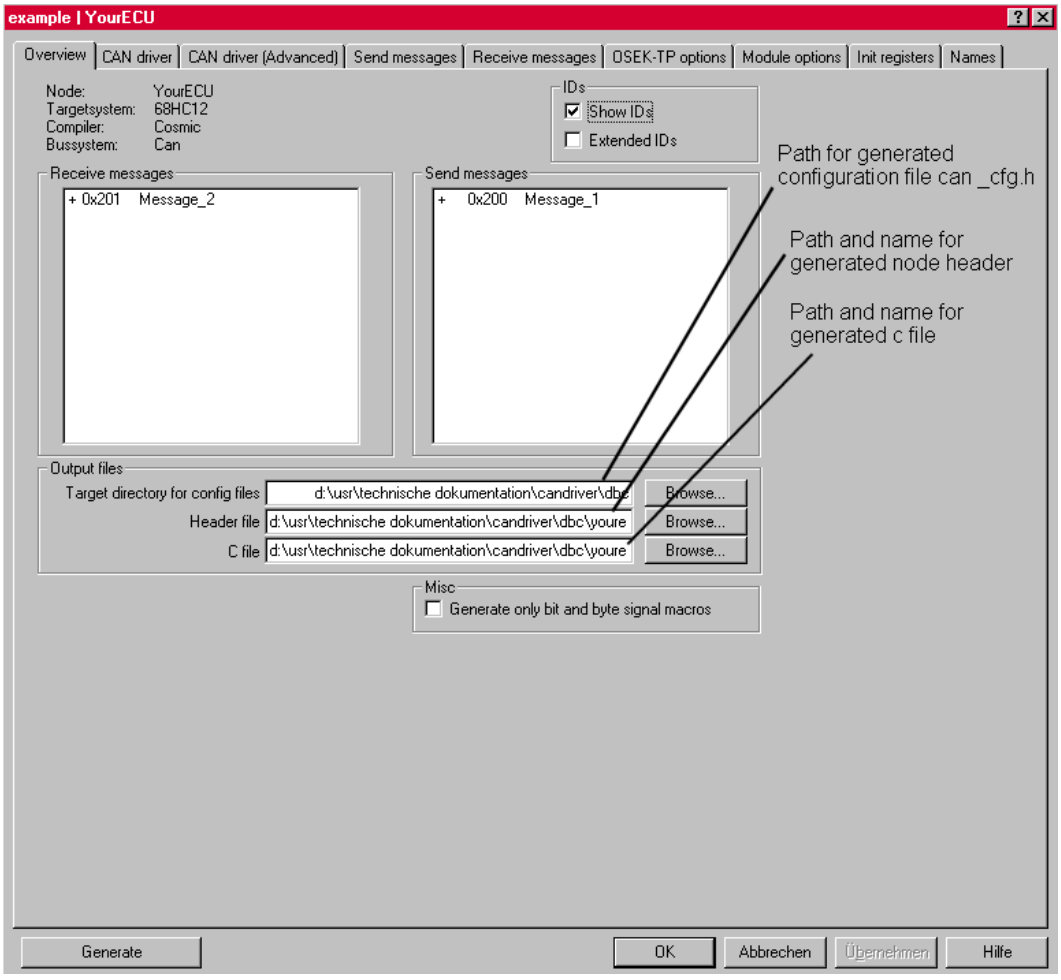


Figure 6-5 Overview of Signals and Directories

In the overview tab page (Figure 6-5) you see all receive and send messages for your node.

Just choose the path where the header and the C file are to be generated and the path for the configuration file can_cfg.h.

Select the tab **CAN Driver**.

In the following dialog you can make settings that are the same for all CAN drivers regardless of the hardware platform. For the first attempt we do not select anything.

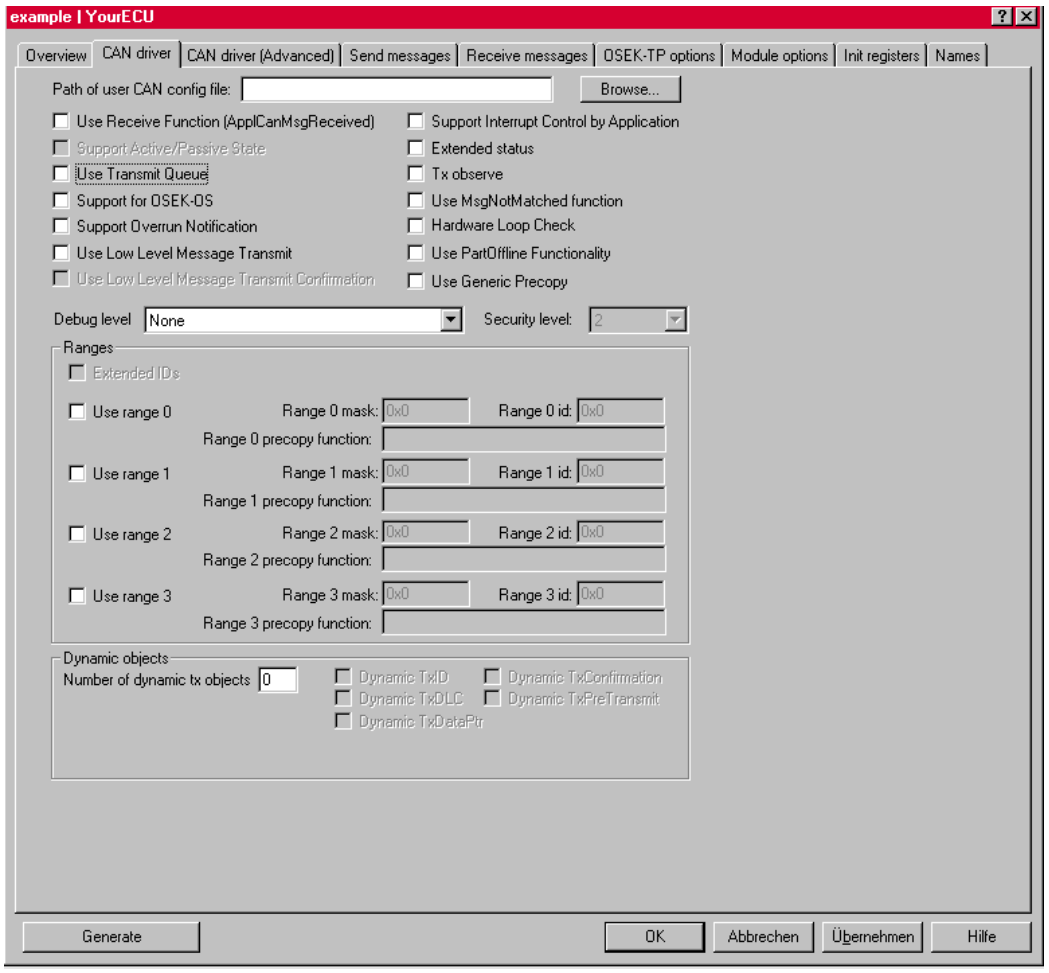


Figure 6-6 CAN Driver Dialog (for HC12)

Here you see the variety of setting you can make to configure a CAN Driver. Take a look at it but do not enter anything for this first attempt.

Refer to the document CANdrv.doc to get further information about these settings.

The next tab, **CAN driver (advanced)** is very special for the specific vehicle manufacturer and the hardware platform. Please refer to the description of your CAN Driver Technical Reference for the advanced settings (TechnicalReference_CAN_**hardwareplatform**.pdf).

Finally we have to set the **Init registers**. Please select this tab (Figure 6-7).

Some of the following information might be hardware dependent but the fundamental mechanisms are the same.

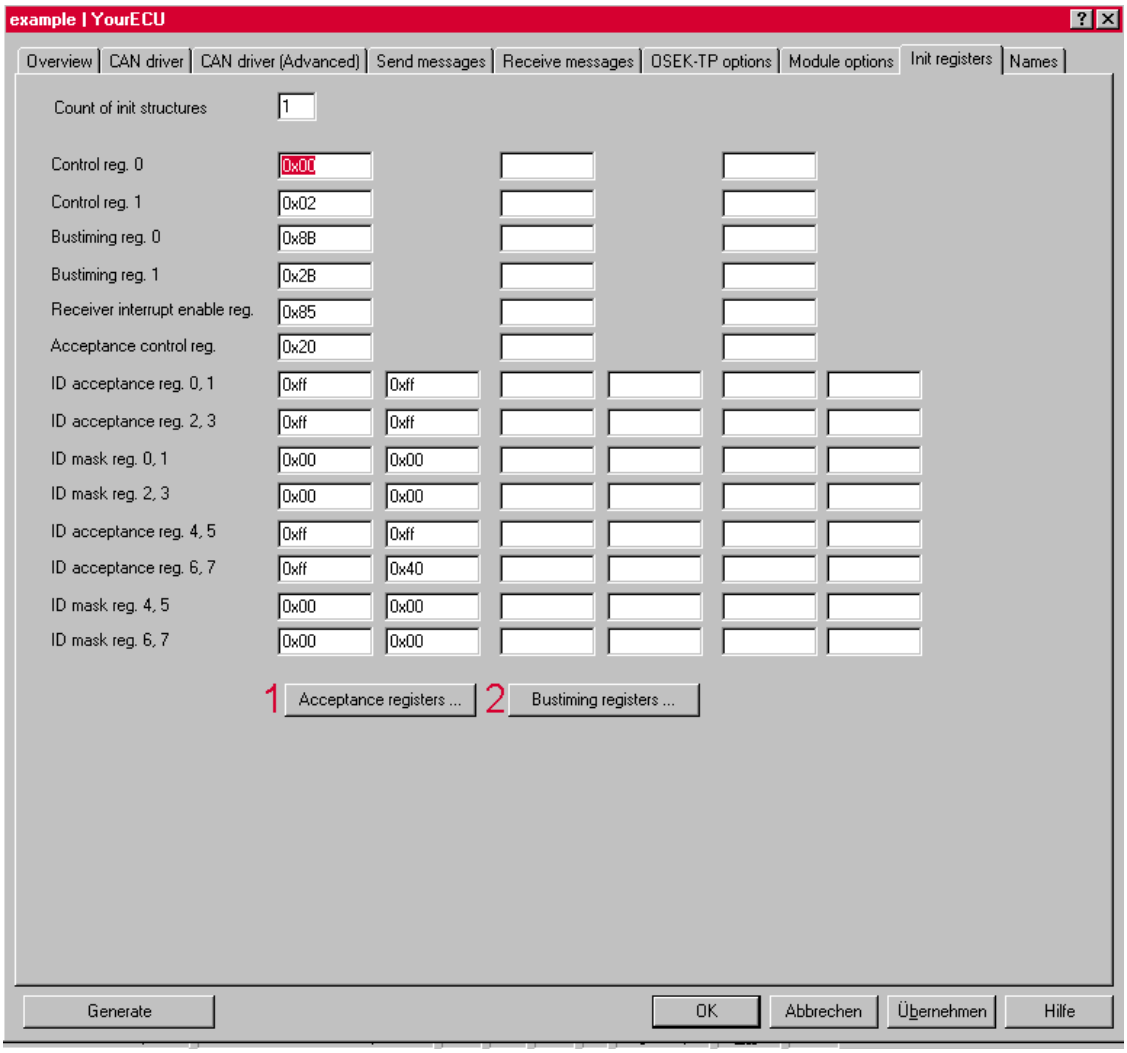


Figure 6-7 Init Registers For The CAN Controller (for HC12)

Every two columns belong to one so-called init structure consisting of baud rate, acceptance filters…

This is a very important dialog. Please pay special attention to these settings.

Here is where you can make the settings for the hardware acceptance filters and the bus timing of your CAN Controller.

First we look at the **Acceptance filters**, second at the **Bustiming registers**.

In order to minimize the number of messages that should not be received by your ECU, you can set a hardware filter by means of the acceptance register (**1**) (see Figure 6-8).

The reception of any message normally causes an interrupt. To minimize the interrupt load you set the filters, so only relevant message will pass and cause an interrupt.

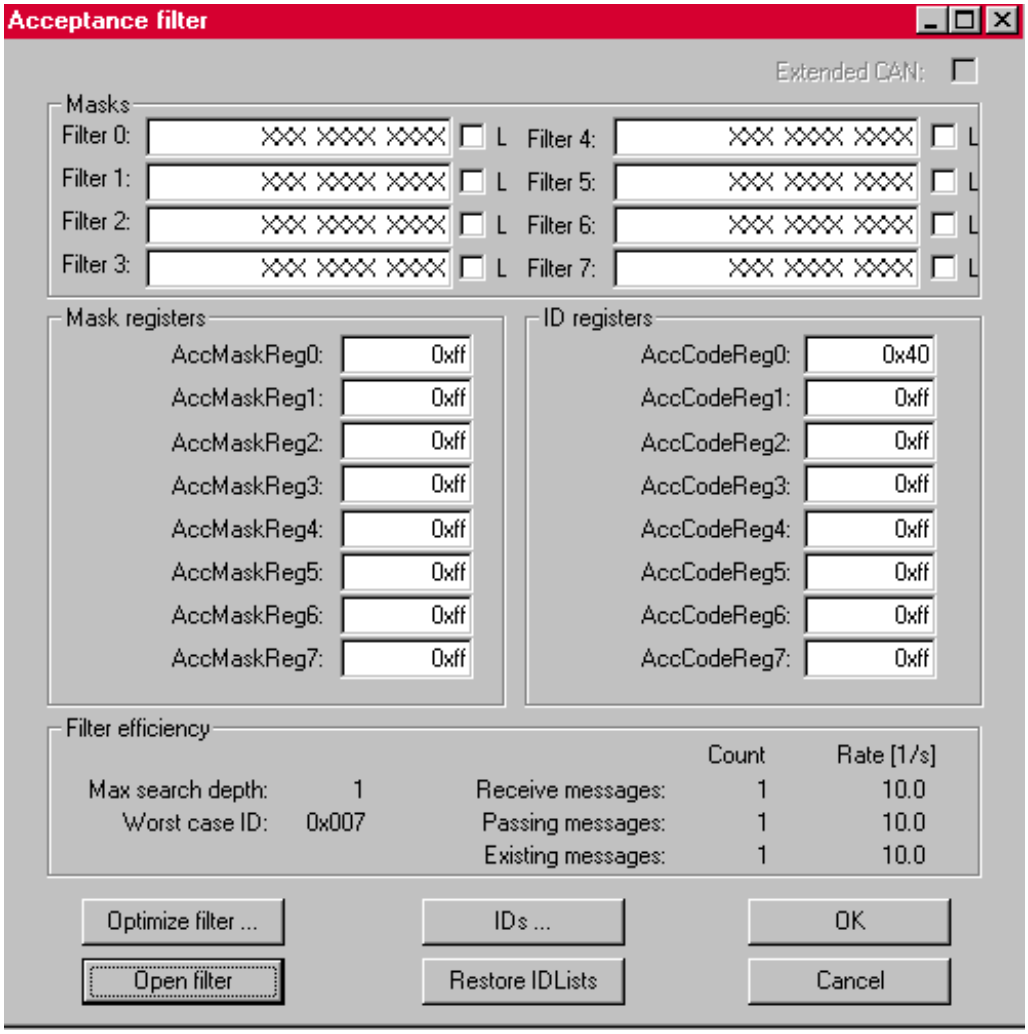


Figure 6-8 Acceptance Filters For The CAN Controller (for HC12)

For the first attempt it is your choice whether to open all filter via the **Open filters** or you use the **Optimize filters** button. Confirm with OK.

The setting of the filters is described in detail in the help document for the Generation Tool. (just use the HELP button).

Next we will look at the **bus timing**. To do this, while still on the **Init registers** tab page, click on the **Bustiming registers** button.

Incorrect Bus Timing settings are common mistakes that cause errors while transmission and reception. Please pay special attention to all settings in this dialog.

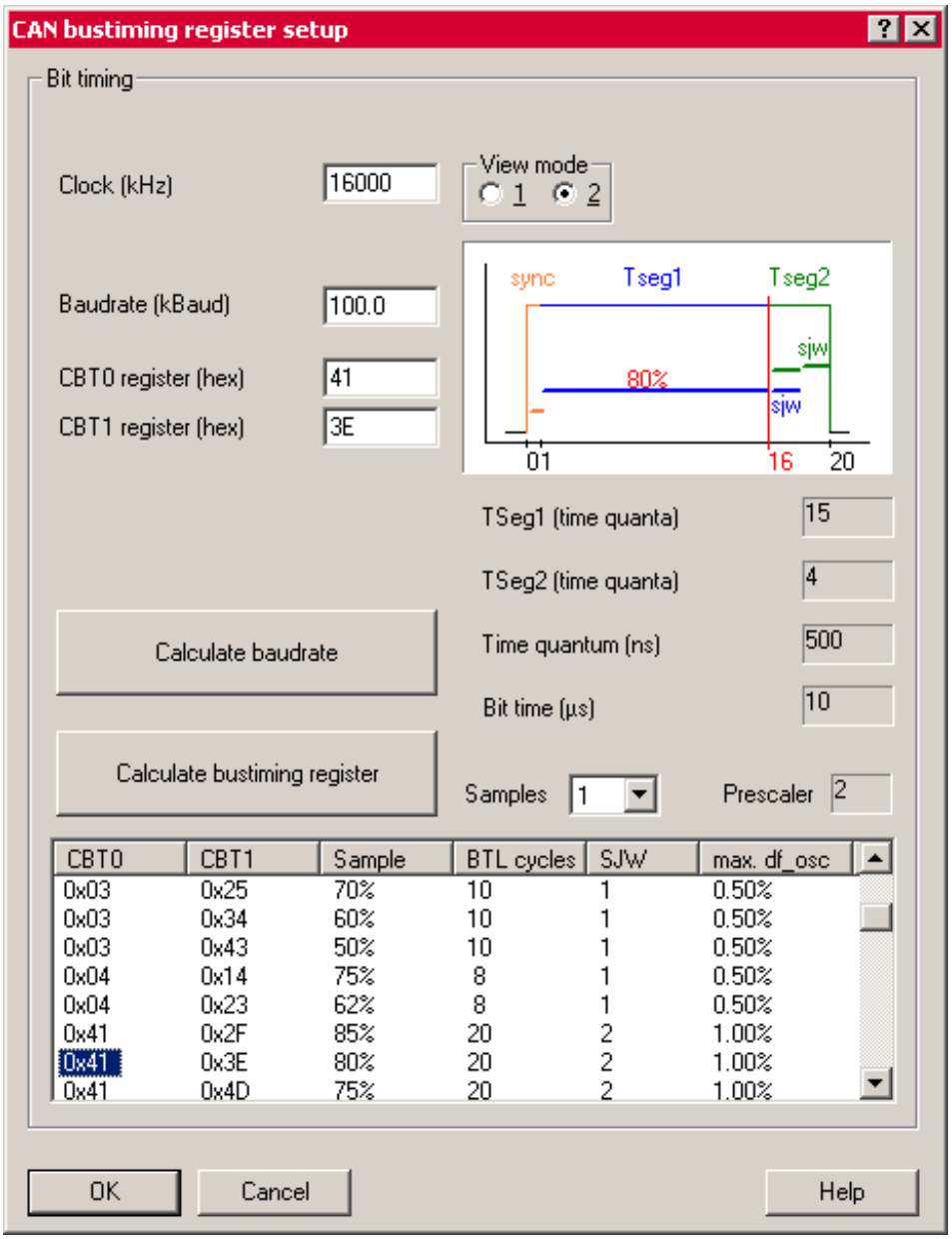


Figure 6-9 Bus Timing Register Settings

Caution

Before SOP it is duty of the OEM / Tier1 supplier to recalculate and validate these automatically calculated values for the bus timing registers.

First you have to enter the clock signal frequency. This is the base for further calculating the timing registers. Make sure to select the correct frequency.

The Bus Timing Registers of any CAN controller contain information about the bus rate, the synchronization jump width (SJW) and the BTL cycles. There are two ways to make these setting:

1. **Do you know the baud rate ?**
   Enter the baud rate and click on **Calculate bustiming registers**. You will get a list of possible register setting. Choose your setting by a click in the list.

Between 60 and 80% is a good value for the Sample Rate and the SJW for your selection. All vehicle manufacturers have strict guidelines for these settings.

2. You can also simply enter values for the two bus timing registers (CBT0 and CBT1) and let the software calculate the baud rate.

Return to the **Init registers** dialog via **OK** and see the changed values.

With a further **OK** you return to the main window of the Generation Tool.

Make sure that the checkboxes **Use TP, Use diagnosis, Use Can Calibration Protocol, Use MCNet …** are NOT selected (as we are working with the CAN Driver only for this example).

The variety of these buttons is dependent on the manufacturer. Deactivate any component but the CAN Driver.

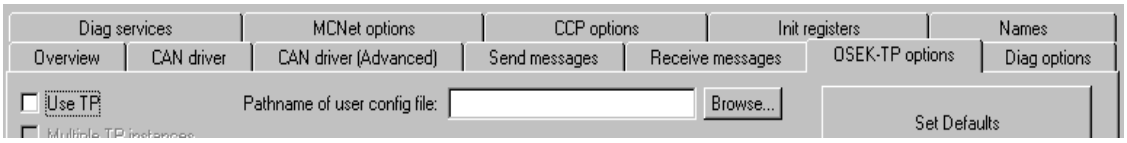


Figure 6-10 TP Options

The figure is only an example. The screenshot may look very different in your case. But you see the checkbox which must not be selected.

Read the following chapter if you use GENy.

6.2.2 Using GENy, the new Generation Tool

The following first steps with the Generation Tool are described in details in the online help of the Generation Tool GENy, too.

Start the Generation Tool and setup a new configuration. Via **File/New** you open the Setup Dialog Window. Select License, Compiler, Micro and Derivative (if available) and confirm via **[OK]**. Then open the Channel Setup Window via the green plus and the selection of the underlying bus system (CAN or LIN).

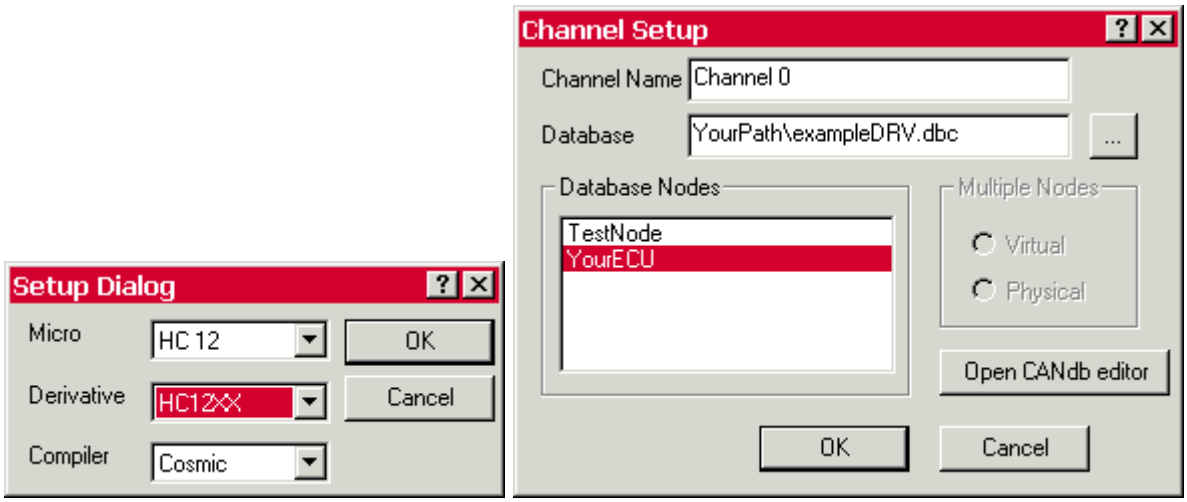


Figure 6-11 Setup Dialog Window and Channel Setup Window to Create a New Configuration

The channel name is Channel X per default (X is starting with 1). Use the browse functionality to enter the location of the data base (dbc file).

Select your node out of the field **Database Nodes** and confirm the settings with **[OK]**.

Now save the configuration via **File/Save** or **File/Save as**.

First at all you should switch on/off the components you need, in this case we only use the CAN Driver.

Use the component selection at the bottom of the main window of the Generation Tool and select the suitable Driver (in this example application we use the CPUHC12 and the Driver HC12).

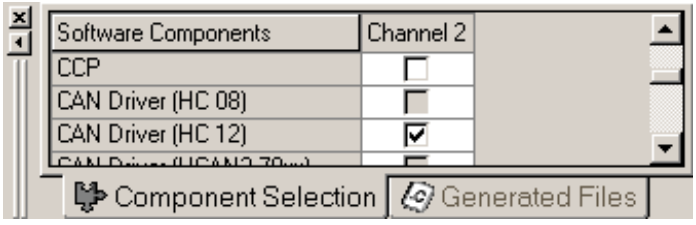


Figure 6-12 Component Selection

Now you should set the path where the Generation Tool generated the files to. To do this, open the Generation Directories Window via **Configuration/Generation Paths**. Enter the root path and select additionally individual paths for the components, if the Generation Tool should generated the files to different folders. This is also described very detailed in the GENy online help.

For the HC12 there are some hardware-specific settings that you have to make, the register block address and the register block offset. It depends on you hardware whether you have to do such kind of settings or not. Refer to the Technical-Reference_CAN_YourHardware.pdf for more information.

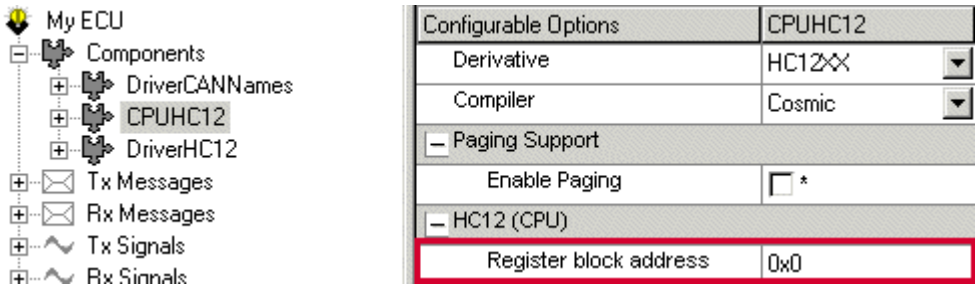


Figure 6-13 The Register Block Address is a General Setting for the CPU

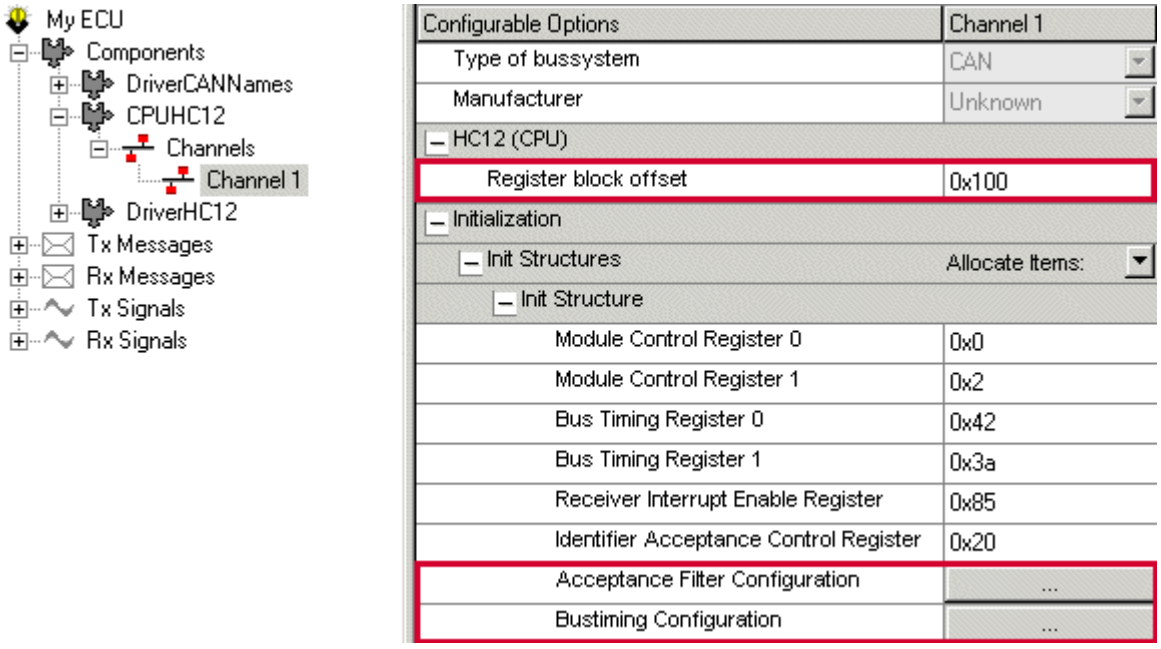


Figure 6-14 Register Block Offset, Acceptance Filters and Bus Timing are Channel-Specific Settings for the CPU

What is missing now is the settings for the acceptance filters and the bus timing.

> Acceptance filter configuration and bus timing configuration are very important settings. Please pay special attention to them.

As you see in the navigation tree above you can open the configuration windows for these two settings via the channels of the hardware (e.g. CPUHC12).

First we take a look at the **Acceptance filters**, second at the **Bustiming registers**.

In order to minimize the number of messages that should not be received by your ECU, you can set a hardware filter by means of the acceptance register.

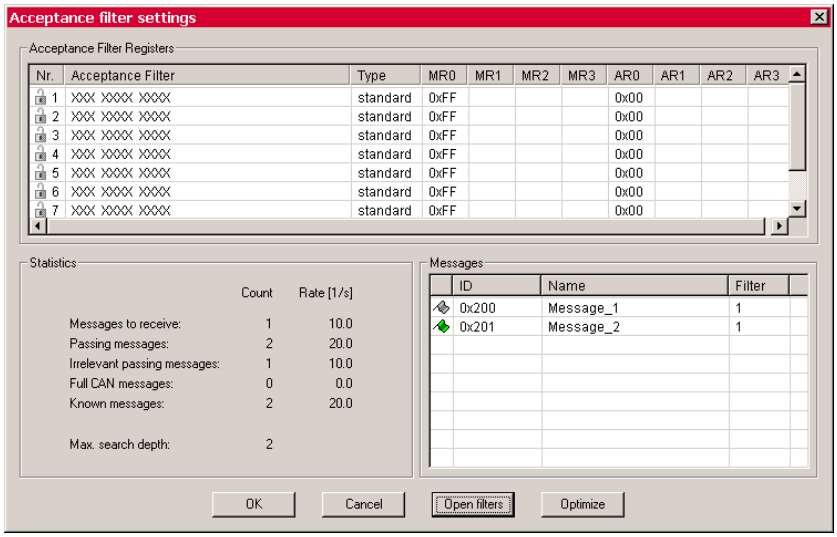


Figure 6-15 Acceptance Filter Settings Window of GENy

The reception of any message normally causes an interrupt. To minimize the interrupt load you set the filters, so only relevant message will pass and cause an interrupt.

For the first attempt it is your choice whether to open all filter via the **Open filters** or you use the **Optimize filters** button. Confirm with **[OK]**.

The window for the bus timing registers is the same as for the CANgen Generation Tool. Refer to the lines above for this explanation.

To make the settings for the component CAN Driver itself use the lists below **DriverHC12** and **DriverHC12/Channels/Channel 1**. But for the this first attempt leave these driver settings on their default values.

Back to 9 Steps overview

6.3 STEP 3 Generate Files

6.3.1 Using CANgen Generation Tool

Click on the button  and start the generation process.

Remember to start the generation process after any change in the dialog windows of the Generation Tool.

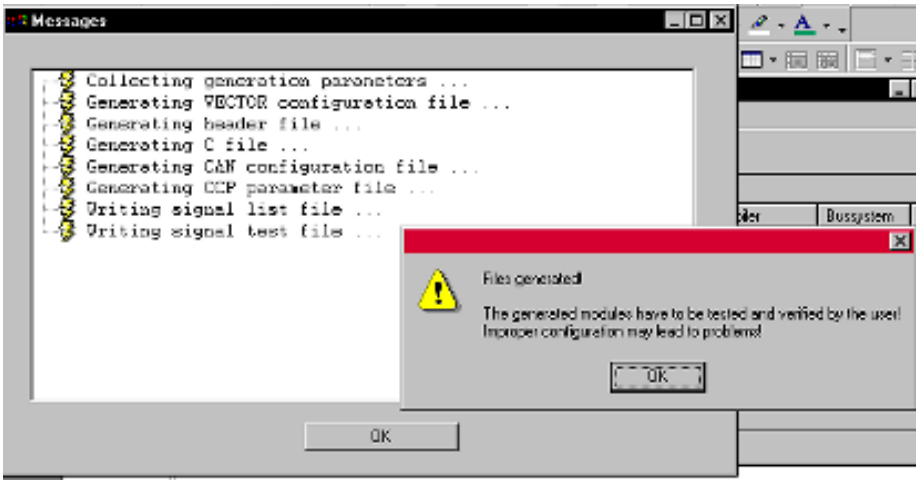


Figure 6-16 Generation Process

The double arrow is only available if you have a multi-channel CAN Driver distinguished via the Channel index.

Now we have generated for the first time. Check the directory and see the new files. There should be at least the files YourECU.c and YourECU.h, and in the path for the configuration file there should be can_cfg.h and v_cfg.h.

If you do not find the generated files check your path in the **Overview** dialog.

6.3.2 Using GENy

Click on the button  and start the generation process.

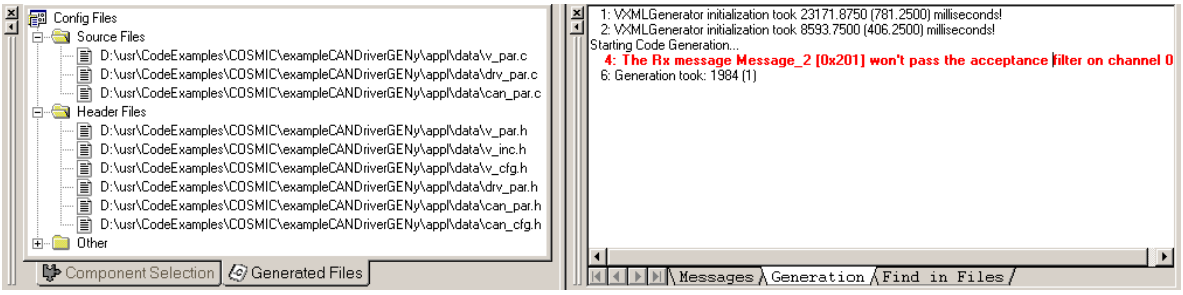


Figure 6-17 Information About the Generated Files and the Generation Process

The Generation Tool provides you with information about the **Generated Files** and **Generation** information. In this example shown above the acceptance filters are not set correctly, Message_2 will not pass the filter. Open or optimize the filters.

If a message will not pass the acceptance filters, the Generation Tool will not create signal access macros for the hardware (_CAN_ see in chapter 5.2.1). Make sure that the generation process runs without error messages.

Now we have generated for the first time. Check the directory and see the new files: can_par.c, can_par.h, drv_par.c, drv_par.h, can_cfg.h, v_par.c, v_par.h, v_cfg.h, v_inc.h.

If you do not find the files check the paths in the **Generation Paths…** dialog.

Back to 9 Steps overview

6.4 STEP 4 Add Files to your Application

In this step you have to add all files of the CANbedded Software Components (In this example these files are only the ones for the CAN Driver) and the generated ones to your project (or makefile).

Rename the file _can_inc.h to can_inc.h (remove the underscore prefix) and open it with an appropriate editor. As you do not use any other component but the CAN Driver you should delete the #include of the Network Management (nm_cfg.h) at the end of this file.

If you want to use functions of the CAN Driver or you want to access signals or messages, you only have to include the `can_inc.h` and then the `YourECU.h` for **CANgen** and `v_inc.h` for using **GENy**.

Now add the driver files to your source list for your compiler or your makefile.

If you want to apply changes you have made in the Generation Tool, you must start the generation process again. Remember compiling afterwards.

!!! We are still not able to compile and link. !!!

The starting point for this example is a very simple application consisting of only one file (here applmain) and an interrupt vector table (vectors.c). It should give you an idea of how to handle the service- and callback functions of the CAN Driver.

In the following chapters we complete this example step by step.

This example was created for using CANgen. For the usage of GENy you just have to include the v_inc.h.

Example for HC12:

```
|-----------------------------------------------------------------------
|               A U T H O R   I D E N T I T Y
|-----------------------------------------------------------------------
| Initials     Name                      Company
| --------     ----------------------    --------------------------
| em           Emmert Klaus              Vector Informatik GmbH
|-----------------------------------------------------------------------

/*Includes  ******************************************************/
#include "can_inc.h"
#include "yourecu.h"

void main (void )
{
}
```

Back to 9 Steps overview

6.5 STEP 5 Adaptations for your Application

To be able to compile and link, you have to adapt a few things in your application.

Example for the HC12:

```
/* Includes*****************************************************/
#include "can_inc.h"  /*using CANgen*/
#include "yourecu.h"  /*using CANgen*/
#include "can_par.h"  /*is also included for GENy via include of v_inc.h*/


/*Function prototypes *****************************************/
void enableInterrupts( void );
void hardwareInit( void );

/*Main Function ***********************************************/
void main(void)
{
/* make sure that the interrupts are disabled to initialize the
   modules.*/
```

DO **NOT** USE any CAN API function **before calling CanInitPowerOn.** **It is forbidden to use CanInterruptDisable here**

```
  hardwareInit();

  CanInitPowerOn(kCanInitObj1);

  enableInterrupts();

  for(;;)
  {
    ;
  }
}

void ApplCanBusOff( void )
{
  ; /*Callback function for notification of BusOff*/
}

void ApplCanWakeUp( void )
{
  ;  /*Callback  function  at  the  transition  from  SleepMode  to  sleep
indication recommended*/
}

void hardwareInit( void )
{
  /*
  Do your hardware specific initializations here.
```

It is forbidden to use any CAN functionality before CanInit-PowerOn !!!

```
  Remember your TRANSCEIVER
  */
  ;
}

@interrupt void irq_dummy0(void)
{
  for( ;; ); /*all other interrupts except the CAN Interrupts are routed
to this function.*/
}
```

Now the description of the above code:

First, we have to include the **`can_inc.h`** and then the generated header, here **`yourecu.h`**.

The following define is to enable the interrupts and is hardware dependent.

In the main() function you have to do initializations first.

In the **`hardwareInit`** you can turn on timers or PWM or something else.

As you see, the transceiver connects directly to the CAN Bus, so:
!!! **PLEASE THINK OF SWITCHING YOUR TRANSCEIVER ON** !!!
THE CAN DRIVER DOES NOT HANDLE THE TRANSCEIVER

Normally this is only necessary when using a low-speed-transceiver. Refer to your hardware guide.

When you use a high speed transceiver, you have to take care of your terminating resistor of 120Ω

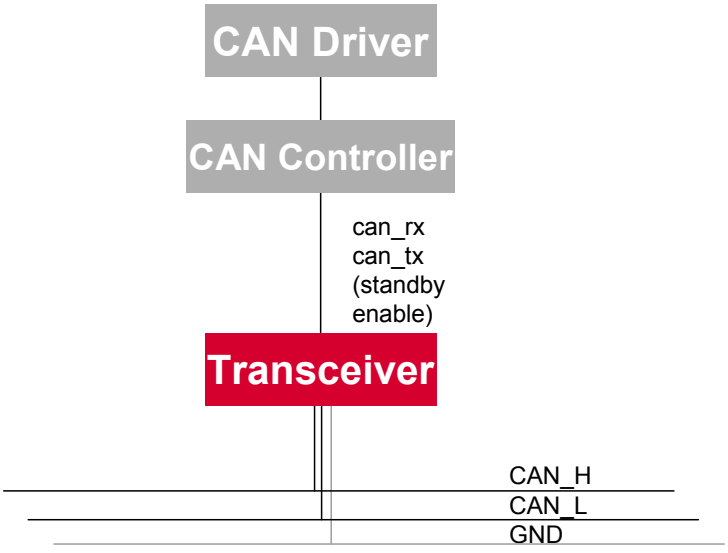


Figure 6-18 The Transceiver

To start the CAN Controller and the CAN Driver, you have to call the function:

`CanInitPowerOn( kCanInitObj1 )`

Make sure that you use functions of the CAN Driver API **after** CanInitPowerOn.

Make sure that the interrupts are disabled when calling the function CanInitPowerOn. Normally the interrupts are disabled by default at startup.

The parameter passed in determines the init structure you made the settings for via the Generation Tool. You will find the macro **kCanInitObj1** in the generated **yourecu.h** (can_par.h for GENy) normally at the end of the file. (On some hardware platforms, this parameter is not necessary (e.g. V850). Refer to your hardware specific documentation for this information.

Now you can enable interrupts.

The main-function is an endless-loop. Perhaps you can turn on some LEDs, to see the application running.

You also have to provide the CAN Driver with two application functions, **applCanBusOff** and **applCanWakeUp**. For the first start, you can leave these functions empty to avoid linker errors.

Think of adding this or a similar file to your system, i.e. your makefile or your project.

Since the CAN Driver uses interrupts for notifying the application when a CAN message has been received, you have to map the interrupts on the corresponding interrupt service routines. Refer to your compiler manual how to do this in your case.

For the HC12 CAN Drivers this is done in the file vectors.c. Let's have a look at this file.

> Interrupts and interrupt tables are highly dependent on the hardware. Just use this example as a guide.

Example for HC12:

```
const functptr vectab[] = {          // @0xFFC4 start address of table
    CanTxInterrupt,                  // $FFC4    CAN transmit
    CanRxInterrupt,                  // $FFC6    CAN receive
    CanErrorInterrupt,               // $FFC8    CAN error
    irq_dummy0,                      // reserved
    irq_dummy0,                      //
    irq_dummy0,                      //
    CanWakeUpInterrupt,              // $FFD0 CAN wake-up
    irq_dummy0,                      //ATD
    irq_dummy0,                      //SCI 2
    irq_dummy0,                      //SPI
    irq_dummy0,                      //SPI
    irq_dummy0,                      //Pulse acc input
    irq_dummy0,                      //Pulse acc overf
    irq_dummy0,                      //Timer overf
    irq_dummy0,                      //Timer channel 7
    irq_dummy0,                      //Timer channel 6
    irq_dummy0,                      //Timer channel 5
    irq_dummy0,                      //Timer channel 4
    irq_dummy0,                      //Timer channel 3
    irq_dummy0,                      //Timer channel 2
    irq_dummy0,                      //Timer channel 1
    irq_dummy0,                      //Timer channel 0
    irq_dummy0,                      //Real time
    irq_dummy0,                      //IRQ
```

```
    irq_dummy0,                     //XIRQ
    irq_dummy0,                     //SWI
    irq_dummy0,                     //illegal
    irq_dummy0,                     //cop fail
    irq_dummy0,                     //clock fail
    _stext                          //RESET
    };
```

As you see in the example, we only use CAN-specific interrupts and the reset vector. All other interrupts result in a dummy interrupt.

You also have to provide the function irq_dummy0( ) in your application. Refer to your hardware description to figure out the solution for your situation.

Back to 9 Steps overview

6.6 STEP 6 Compile, Link and Download

Now start your compiler by calling the **makefile** or just clicking the start button. What you do is depends on your development tool chain.

Back to 9 Steps overview

6.7 STEP 7 Receiving A Message

Congratulations !!

You have now arrived at **Step 7**, i.e. you can compile and link. You are very close to your first CAN communication.

In every project you normally have to spend a lot of the time starting up the hardware and the development environment.

Make sure that:
You have the correct clock frequency (important for baud rate).
You have entered this clock in the dialog box of the Generation Tool.
The memory mapping is correct.
The physical CAN connection is there and has no damage.
You have a terminating resistor (120Ω) if you use high-speed CAN (powertrain).
Your transceiver is initialized properly

After the download of your Application, set a breakpoint in your debugger on the main (void) function and start. Did it stop at main?

If so, **Congratulations** again.

Remove the breakpoint and restart. Now your application is running in the endless loop. Please check this!

Figure 6-19 Simple Test Environment

CANoe is very convenient for testing a CAN communication.

As soon as you see the message Id or the Name in the Trace window after sending a message, you know that your hardware settings are correct.

Now send a CAN message on the bus. It is best to use an ID your ECU normally has to receive.

A very easy way to send a message is by using the CANoe (or CANalyzer), a PC-tool from Vector Informatik. Use the generator block and send the message.

Send the message and observe the Trace window.

Do you see the name of the message or the ID?

Great, your ECU has acknowledged the CAN message, i.e. all hardware settings are correct.

If you see Error frames, check the list above. The main mistakes are hardware settings (transceiver), the baud rate (clock of CAN Controller), wiring problems and ground offsets.

Now we are ready to modify our application. Please check in the Generation Tool on the tab **Receive messages**, if the **Indication flag** for the message is switched on.

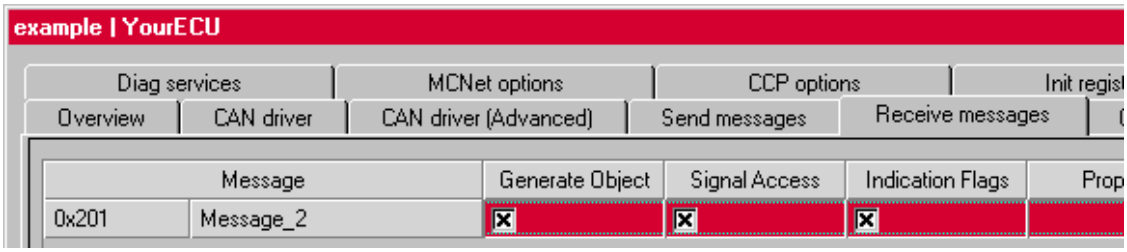


Figure 6-20 Check button for indication flag

If not, switch it on, start a new generation process, compile and link the system again and download it to the target.

There are many more ways to react when a CAN message is received. Refer to Step 9 in this manual.

Now we use the so-called **Indication flag** to poll the reception of the CAN message. When an interrupt is triggered by the reception of a CAN message, the indication flag will be set (if chosen in the Generation Tool).

When you use another dbc file, your message will have a different name. You will find the correct macro for your indication flag in the file yourecu.h (can_par.h). Just search for **indication**.

Example for the HC12:

```
void main(void)
{
  hardwareInit();

  CanInitPowerOn(kCanInitObj1);

  enableInterrupts();

  for(;;)
  {
    /* First Test modification*/
    if( Message_2_ind_b )
    {
      Message_2_ind_b = 0; Breakpoint here
```

```
        }
    }
}
```

The names are generated out of the name of the signal and the pre- and postfixes from the “names” tab. Refer to the file YourECU.h (can_par.h) for the correct writing of messages, indication flags etc.

Compile, link and download the application and set a breakpoint (as shown). Now send the CAN message via the Generator Block.

It stopped at the breakpoint?

If so, you received the message, used the correct macro for the flag and got notified of the reception.

Back to 9 Steps overview

6.8 STEP 8 Sending a Message

The next step is the transmission of a CAN message. This is done by calling the function

`CanTransmit( handle );`

The **`handle`** specifies the message you want to send. Open the file **`yourecu.h`** (`can_par.h` for GENy) and search for "**`handle`**" in the section on send/transmit objects. Chose the appropriate macro for the send message and use it as shown.

Example:

```
void main(void)
{
  hardwareInit();

  CanInitPowerOn(kCanInitObj1);

  enableInterrupts();

  for(;;)
  {
    /* First Test modification*/
    if( Message_2_ind_b )
    {
  /*Second Test modification*/
      if( CanTransmit( CanTxMessage_1 ) == kCanTxOk )
  {
    Message_2_ind_b = 0;
  }
    }
  }
}
```

handle

Refer to the file YourECU.h (can_par.h for GENy) for the message handles.

Meaning of the return value of CanTransmit

The function CanTransmit has a return value, `kCanTxOk` or `kCanTxFailed`. The meaning of this value is not as simple as it looks like.

Without Software Transmit Queue

`kCanTxOk` means that the data has been copied from the RAM to the Tx Register and the transmit request is set in the CAN Controller hardware. The physical transmission of the message depends on when the message wins the CAN bus arbitration.

`kCanTxFailed` means the hardware register is busy or CAN Driver is offline.

With Software Transmit Queue

kCanTxOk could mean the same as above. But it also could mean that the transmit request is set in the software queue of the CAN Driver and will be processed as soon as possible. Read more about the software queue in the chapter 6.9.2.3.

kCanTxFailed means the CAN Driver is offline.

Full CAN Tx Object

There is no Tx queue functionality for Full CAN Tx Objects.

This modified application sends back a CAN message. You should see the response message in your Trace window. Refer to the TechnicalReference_CANDriver.pdf to get information about the return value.

Do you see the response message in the Trace window?

CONGRATULATIONS!

The basic CAN communication is running.

Of course this is a very simple application and far away from the complexity of your application, but as the saying goes:

The longest way starts with the first step(s).

Back to 9 Steps overview

6.9 STEP 9 Further Actions

The next step is to build up your application around the communication. To do this in a simple manner, read the following tips and recommendations. You will then be given an exercise that poses a problem which you will try to solve. After finding the correct answer, you will understand the order in which the functions of the CAN Driver are called.

6.9.1 Strategies for Receiving a CAN Message

The Figure 6-21 shows the calling order of the functions when receiving a message.

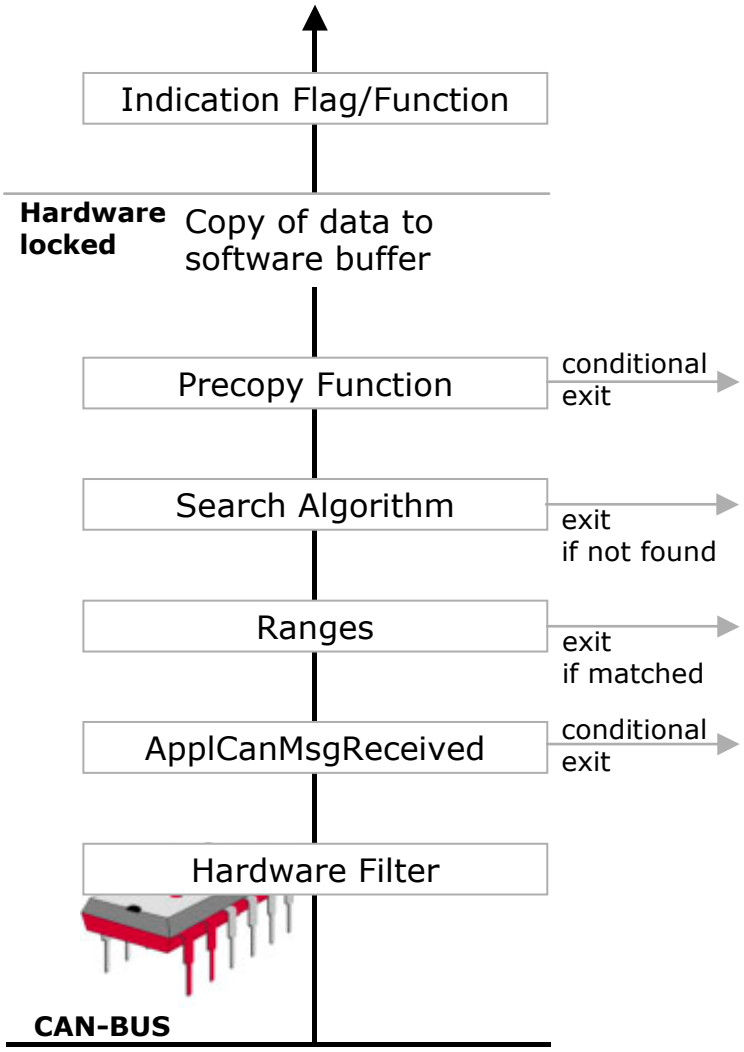


Figure 6-21 Calling Order Of Functions When A CAN Message Is Received

6.9.1.1 Hardware Filter (HW Filter)

As you have learned before, you can adjust the hardware filter in the Generation Tool. Every message that passes the hardware filter triggers an interrupt – if not using polling mode.

To reduce your interrupt load, optimize the filters (Generation Tool).

6.9.1.2 ApplCanMsgReceived

In the Generation Tool you can choose to have this function called with any reception of a CAN message that passes the hardware filter.

Here you can filter the messages that pass the hardware filter but contain no relevant information. At this point, the data is in the receive register (**Rx register**). Use the hardware access macros to figure out ID, DLC and DATA of the received message.

If you select the function ApplCanMsgReceived, the prototype will be generated. You only have to enter the function.

6.9.1.3 Ranges

Ranges are a software filter mechanism. Since the message is filtered by its ID and assigned to the categories Network Management, Diagnostics, Application, etc., you can route more messages on the same **`Range precopy function`**.

6.9.1.4 Search Algorithm

To figure out whether a message is meant for your ECU and which message ID it is the CAN Driver has to compare the ID with an ID-list. This comparison can be done in different ways. The criterion is the speed for browsing the list. The Generation Tool offers different search algorithms to choose. Please refer to the CAN Driver Technical Manual for the differences.

6.9.1.5 Precopy

In the Generation Tool (**receive objects/functions**) you can enter a name for a message-specific precopy function. This function is called by the CAN Driver before copying the message data from the receive register to the RAM data structure (if selected).

The _CAN_ macros will be generated only when you have chosen a precopy function or the object is a full can obiect.

The **`Precopy function`** e.g. can be used to check to see if the data has changed. Use the **`_CAN_`** access macros to read the data out of the receive register and compare it with the RAM buffer (look in **`yourECU.h`** for CANgen and in **`can_par.h`** for GENy for these access macros).

This macro will be only generated if the message is a full can object or you have selected a precopy-function for this message (see later).

As you are in interrupt context, data consistency is not in danger (refer to the technical reference for you hardware). Keep your actions short in any **`Precopy function`**.

The return value of the **`Precopy function`** determines what happens next.

`kCanCopyData`: The driver is to do the copying from receive register to RAM buffer.

`kCanNoCopyData`: You did the copy of relevant data and the driver does not need to call its copying routine.

6.9.1.6 Indication Flag / Indication Function

The application is notified by the indication flags and/or the indication functions

The driver indicates the reception of a message to the application.

`Indication Function`
This function runs in interrupt context and is message-specific. Keep your action very short in this function.

You can enter a name for a message-specific **`Indication function`** in the Generation Tool (in the same way as you did it for the **`Precopy function`**).

`Indication Flag`
This flag is message-specific. It is set by the CAN Driver and can be polled by the application. It tells the application a new message has been received.

!!! This flag must be reset by the application. !!!

You can use this flag to ensure you are working on the newest contents of a message.

If the flag is set, it means that a new message has been received. Clear the flag and access the received data. If the flag is cleared after your access, you can be sure that you have current data. If the flag is set, new data has been received in the meantime, so repeat this once again.

Remember for Data consistency

All flags are set by the driver in an interrupt context. If your µC does not perform an atomic (i.e. uninterruptible) write access to bit data it is necessary to protect the write access via an interrupt lock to prevent unexpected change or loss of flag states. It is recommended to perform this (CAN-) interrupt lock via the API functions CanInterruptDisable / CanInterruptRestore.

Please refer to your controller and/or compiler manual for information about atomic write access to bit data in your system. In case of doubts, it is recommended to lock the interrupt during the access.

6.9.2 Strategies for Sending a CAN Message

The transmission of a CAN message is divided into two parts: the preparations until the transmission of a message and the transmit interrupt handling, which informs you about the successful sending of a message.

First we look at the preparations before sending.

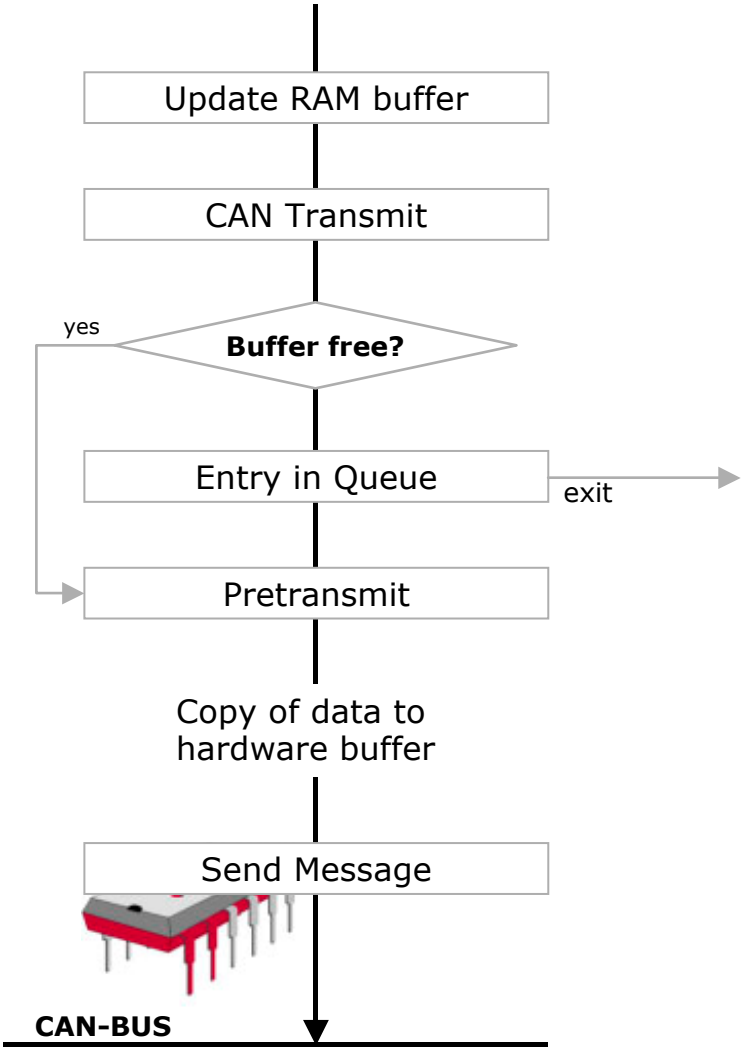


Figure 6-22 States Before Transmitting A CAN Message

6.9.2.1 Update RAM buffer

The methods of transmission highly depend on your application. You may have to send in a fixed cycle, or you may have to send when a specific event occurs.

Both cases require current data. If you use the RAM buffer, you just have to keep the data in it up-to-date.

When you use your own buffer you should enter the values directly into the transmit register just before sending the message. Otherwise the data could be overwritten by another transmit message.

6.9.2.2 CanTransmit

The CanTransmit function initiates the transmission of a specific message. The handle (in the file yourECU.h for CANgen and in can_par.h for GENy) specifies which message is sent.

CanTransmit has 2 possible return values, but none of them guarantee that the message has been sent yet. They just say that either the message will be sent as soon as possible (with or without a queue) or the transmission request has been rejected (see 6.8);

You can only be sure that the message has been sent when you get the confirmation interrupt (confirmation function or confirmation flag)

6.9.2.3 The Queue

There are two possible causes for a return value of kCanTxOk: either the message has directly entered the transmit buffer or the message has entered the queue (if chosen in the Generation Tool).

An entry in the queue means, the **REQUEST** to send this message is stored, not the data. So leave the data untouched until you are sure the message has been sent, i.e. until the **`Confirmation flag`** is set or the **`Confirmation function`** is called.

6.9.2.4 Pretransmit Function

When you do not have a RAM buffer for your message, you must copy the data to the transmit register of the CAN Controller in the **`Pretransmit function`**. With the call of the function you get a pointer directly to the transmit registers. In this case you have to know the distribution of the signals to the bytes, because you do not get signal access macros.

> The time between the call of CanTransmit and the confirmation interrupt is not predictable. To update your data short before the transmission, use the Pretransmit function too. If this function is called, you can be sure that this message is the next message to be sent.

When the message is sent and at least one ECU receives this message, the acknowledge will trigger the so-called transmit interrupt. This interrupt calls the transmit interrupt service routine, which in turn might call the confirmation function or set the confirmation flag.

6.9.2.5 Confirmation Function and Confirmation Flag

The **`Confirmation function`** is called in interrupt context, so keep the run time as short as possible by not doing much in the code. **Now** you can be sure, your message has been sent successfully.

See the parallels between the indication function/flag and the confirmation function/flag

The **`Confirmation flag`** has the same meaning, but you have to poll this flag in your application (similar to the way it is done with the **`Indication flag`**). You get the macro out of the generated file **`yourECU.h (can_par.h)`**.

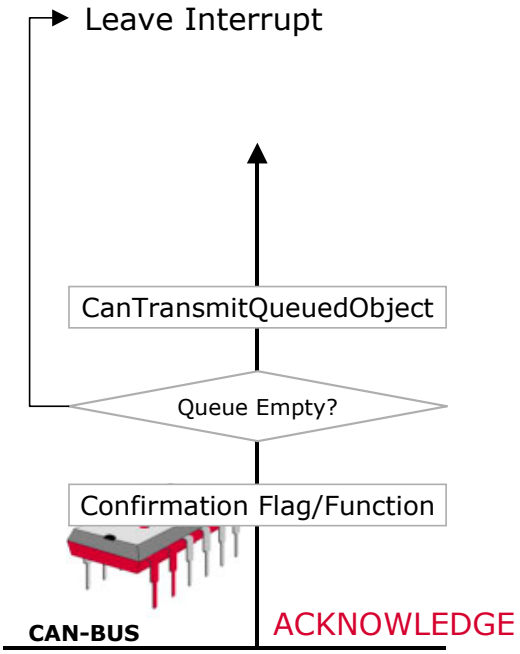


Figure 6-23 Confirmation Interrupt After Transmission Of CAN Message

After the confirmation the queue (if chosen) will be worked on.

Back to 9 Steps overview

7 Further Information

7.1 An Exercise For Practice

The program below is a small application (**for the HC12**) using the basic service- and call-back functions of the CAN driver. Try to follow the program and answer the questions connected with.

The application receives a CAN message containing a byte signal **b_Signal_2_c**.

After processing the incoming message, the application sends another message containing only one byte signal, called **b_Signal_1_c**.

Can you calculate the value that will be sent back with **b_Signal_1_c** depending on the value of the variable **a** and the value of the received signal **b_Signal_2_c**?

See the solution at the end of this document

Example for the HC12:

```
/* Includes
******************************************************************/
#include "can_inc.h"    /*for CANgen*/
#include "yourecu.h"    /*for CANgen*/

#include "can_par.h"    /*only include for GENy*/

/* Variable definition
******************************************************************/
unsigned char a;


/* Function prototypes
*****************************************************/
void main_function(void);
void enableInterrupts( void );
void hardwareInit( void );

/* The Main Function
*****************************************************/

void main(void)
{
  hardwareInit();

  CanInitPowerOn(kCanInitObj1);

  a = 5;
  b_Signal_2_c = 0;
```

```
  enableInterrupts();

  for(;;)
  {

    if( Message_2_ind_b )
    {
      b_Signal_1_c = b_Signal_2_c + a;
      if( CanTransmit( CanTxMessage_1 ) == kCanTxOk )
      {
        /*Disable Interrupt*/
        Message_2_ind_b = 0;
        /*Enable Interrupt*/
      }

    }
    if( Message_1_conf_b )
    {
      a=1;
                  /*Disable Interrupt*/
      Message_1_conf_b = 0;
                  /*Enable Interrupt*/
    }

  }
}

/* Other Functions
************************************************************/

void enableInterrupts( void )
{
  /*Enable interrupts*/
}
void ApplCanBusOff( void ) /*later uesd for bus off treatment*/
{
  ;
}
void ApplCanWakeUp( void ) /*later used for wakeup functionality*/
{
  ;
}
/* new function added in the Generation Tool and application */
canuint8 ApplCanMsgReceived( void )
{
  a=a+2;
  return( kCanCopyData );
}
canuint8 M1_PretransmitFunction(CanChipDataPtr dat)
{
  b_Signal_1_c = b_Signal_1_c +1;
  return( kCanCopyData );
}
void M1_ConfirmationFunction(CanTransmitHandle tmtObject)
{
  a=a+23;
}
canuint8 M2_Precopy(CanReceiveHandle rcvObject)
{
```

```
      rcvObject = rcvObject;  /*to  avoid  compiler  warning.  You  only
                                use  this  handle  if  you  have  one
                                precopy  function  for  two  or  more
                                messages*/
  a=2;
  if( b_CAN_Signal_2_c == b_Signal_2_c )
  {
    return( kCanNoCopyData ); /*same  value  as  before,  no  need  to
                                    copy data, leave interrupt*/
  }
  else
  {
    a = b_CAN_Signal_2_c;
    return( kCanCopyData );
  }
}

void M2_IndicationFunction(CanReceiveHandle rcvObject)
{
  rcvObject = rcvObject; /*to  avoid  compiler  warning.  You  only
            use      this      handle      if      you      have      one
                              precopy  function  for  two  or  more
                              messages*/
  a=a+1;
}
/*****************************************************/
void hardwareInit( void )
{
  /*
  Do your hardware specific initializations here.
  Don't forget to initialize your TRANSCEIVER, if necessary*/
  ;
}
@interrupt void irq_dummy0(void)
{
  for( ;; );
}
```

7.2 The Solution To The Exercise

7.2.1 After the first reception and transmission of a new value:

```
a = 1
b_Signal_1_c = b_Signal_2_c*2+2
```

7.2.2 After the reception of the same value as before:

```
a = 2
no return message
```

Did you get it?

7.2.3 The solution, step by step

At the beginning,

a=5 and b_Signal_2_c = 0

The Main loop is waiting on an Indication Flag for Message 2 and a Confirmation Flag for Message 1.

The first function that is called after the reception of Message 2 is **`ApplCanMsgReceived`**. There the **2** is added to **a** and returned with **kCanCopyData**, so the reception process continues.

> At this point a=7 and b_Signal_2_c = b_Signal_2_c

The next function executed is **`M2_Precopy`**. The variable **a** is set to **2** and the received signal b_CAN_Signal_2_c (the data in the Rx register, i.e. in the CAN Controller) is compared with the signal b_Signal_2_c.

If there is no change, the return value kCanNoCopyData stops the receive process the receive interrupt is exited.

> Now a=2 and no response message will be sent.

If there is a difference, **a** is set to b_CAN_Signal_2_c and the return value keeps the receive interrupt going on.

> At this point a= b_CAN_Signal_2_c and b_Signal_2_c= b_Signal_2_c

Now the **`Indication function M2_IndicatinoFunction`** is called, where **a** is increased by 1.

> Now a= b_Signal_2_c+1 and b_Signal_2_c= b_Signal_2_c

Since the **`Indication flag`** is set, now the main loop put b_Signal_1_c = b_Signal_2_c+a, i.e. b_Signal_1_c= 2* b_Signal_2_c+1.

Now the message is requested to be sent (CanTransmit). If the request is answered with a kCanTxOk, the Indication flag is cleared to avoid sending the message again and again. Otherwise the flags stay set until the request of sending this message is successful.

At this point a = b_Signal_2_c+1 and b_Signal_2_c = b_Signal_2_c and b_Signal_1_c = 2* b_Signal_2_c +1

Before the message is on its way, the function **`M1_PretransmitFunction`** is called. This function increments the send signal by 1 and the return value lets the driver copy the data from the RAM buffer to the Tx register.

At this point a = b_Signal_2_c+1 and b_Signal_2_c = b_Signal_2_c and b_Signal_1_c = 2* b_Signal_2_c +2

Now the message is on its way via CAN. The first node to receive this message (in this test case, CANoe) gives an acknowledge that triggers a transmit interrupt.

In the function **`M1_ConfirmationFunction`** 23 is added to **a**.

Now a = b_Signal_2_c+24 and b_Signal_2_c = b_Signal_2_c and b_Signal_1_c = 2* b_Signal_2_c +2

Then the **`Confirmation flag`** is set and recognized by the main loop. There the variable a is set to 1 and the **`Confirmation flag`** is reset to 0 to prevent setting **a** over and over again.

So the result is:

a = 1
b_Signal_1_c = 2* b_Signal_2_c +2

8 Index