

MICROSAR CAN Driver

Technical Reference

Texas Instruments

Tms470 / Tms570

Dcan

Version 1.10.00

Authors	Georg Pflügel, Sebastian Gärtner, Mihai Olariu, Robert Schelkle
Status	Released

1. Document Information

1.1 History

Platforms

Author	Date	Version	Remarks
Georg Pflügel	2007-01-12	1.00	Initial version.
Georg Pflügel	2007-11-15	1.01	CAN-Driver Update to ASR2.1.
Georg Pflügel	2008-08-05	1.02	CAN-Driver Update to ASR3
Sebastian Gärtner	2009-09-11	1.03	CAN-Driver Update to R7.
Sebastian Gärtner	2009-12-14	1.04	Local power-down mode added.
Mihai Olariu	2010-07-28	1.05	CAN-Driver Update to R9. Description for DCAN Issue#22.
Georg Pflügel	2011-03-08	1.06	Description of mailbox objects updated
Georg Pflügel	2011-07-06	1.07	Description for the TMS570LS30316U added
Georg Pflügel	2011-08-24	1.08	Description for GeneratorGeny added
Georg Pflügel	2012-07-19	1.09	Update to R14
Robert Schelkle	2013-01-16	1.10	Update to template 2.05.01 Adapt Overrun/Overwrite description

Table 1-1 History of the Document

1.2 Reference Documents

No.	Title	Version
[1]	AUTOSAR_SWS_CAN_DRIVER.pdf	2.4.6 + 3.0.0 + 4.0.0
[2]	AUTOSAR_BasicSoftwareModules.pdf	V1.0.0
[3]	AUTOSAR_SWS BSW Scheduler	V1.1.0
[4]	AUTOSAR_SWS_CAN_Interface.pdf	3.2.7 + 4.0.0 + 5.0.0
[5]	AN-ISC-8-1118 MICROSAR BSW Compatibility Check	V1.0.0

Table 1-2 Reference Documents

1.3 Scope of the Document

This document describes the functionality, API and configuration of the MICROSAR CAN driver as specified in [1]. The CAN driver is a hardware abstraction layer with a standardized interface to the CAN Interface layer.

**Please note**

We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector's release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

Contents

1. Document Information	2
1.1 History.....	2
1.2 Reference Documents.....	2
1.3 Scope of the Document.....	3
2. Hardware Overview	6
3. Introduction	7
3.1 Architecture Overview	7
4. Functional Description	9
4.1 Features.....	9
4.2 Initialization	13
4.3 Communication	13
4.4 States / Modes	15
4.5 Re-Initialization.....	15
4.6 CAN Interrupt Locking	16
4.7 Main Functions.....	16
4.8 Error Handling	16
5. Integration	21
5.1 Scope of Delivery	21
5.2 Include Structure	22
5.3 Critical Sections	22
5.4 Compiler Abstraction and Memory Mapping	24
5.5 Hardware Specific Hints	25
6. API Description	26
6.1 Interrupt Service Routines provided by CAN	26
6.2 Services provided by CAN.....	28
6.3 Services used by CAN	40
7. Configuration.....	42
7.1 Pre-Compile Parameters	42
7.2 Link-Time Parameters	43
7.3 Post-Build Parameters	43
7.4 Configuration with GENy	44
7.5 Configuration with da Vinci Configurator	54

8. AUTOSAR Standard Compliance	55
8.1 Limitations / Restrictions.....	55
8.2 Vector Extensions	55
9. Glossary and Abbreviations	56
9.1 Glossary.....	56
9.2 Abbreviations	56
10. Contact.....	57

Illustrations

Figure 3-1	AUTOSAR architecture.....	7
Figure 3-2	Interfaces to adjacent modules of the CAN.....	8
Figure 5-1	Include Structure (AUTOSAR)	22
Figure 6-1	Select OS Type.....	26
Figure 7-1	Platform settings	44
Figure 7-2	Init Structure Dialog	52
Figure 7-3	Setup Filter Dialog	53
Figure 7-4	Baud Rate Dialog	54

Tables

Table 1-1	History of the Document	2
Table 1-2	Reference Documents	2
Table 2-1	Supported Hardware Overview	6
Table 4-1	Supported features	12
Table 4-2	Hardware mailbox layout	14
Table 4-3	Errors reported to DET	17
Table 4-4	API from which the Errors are reported.....	17
Table 4-5	Errors reported to DEM.....	18
Table 4-6	Hardware Loop Check	19
Table 5-1	Static files	21
Table 5-2	Generated files	21
Table 5-3	Critical Section Codes	24
Table 5-4	Compiler abstraction and memory mapping.....	25
Table 6-1	Services used by the CAN	41
Table 7-1	Platform Parameter description.....	44
Table 7-2	Controller Parameter description	51
Table 7-3	Filter Parameter description.....	53
Table 7-4	Baud rate Parameter description	54
Table 9-1	Glossary	56
Table 9-2	Abbreviations.....	56

2. Hardware Overview

The following table summarizes information about the CAN Driver. It gives you detailed information about the derivatives and compilers. As very important information the documentations of the hardware manufacturers are listed. The CAN Driver is based upon these documents in the given version.

Derivative	Compiler	Hardware Manufacturer Document	Version
TMS470PSF761	TI	TMS470PSF761 DesignSpec.pdf	Revision 0.8
TMS570PSFC61	TI	TMS570PSFC61_Specification_044.pdf	Revision 0.44
TMS570PSFC66	TI	TMS570PSFC66_design_specification_22.pdf	Revision 2.2
TMS570LS30316U	TI	Gladiator_design_specification_GM_Auto.pdf	Version 2.5.1

Table 2-1 Supported Hardware Overview

Derivative: This can be a single information or a list of derivatives, the CAN Driver can be used on.

Compiler: List of Compilers the CAN Driver is working with

Hardware Manufacturer Document Name: List of hardware documentation the CAN Driver is based on.

Version: To be able to reference to this hardware documentation its version is very important.

3. Introduction

This document describes the functionality, API and configuration of the AUTOSAR BSW module CAN as specified in [1].

Since each hardware platform has its own behavior based on the CAN specifications, the main goal of the CAN driver is to give a standardized interface to support communication over the CAN bus for each platform in the same way. The CAN driver works closely together with the higher layer CAN interface.

Supported AUTOSAR Release*:	3 and 4	
Supported Configuration Variants:	pre-compile, link-time, post-build	
Vendor ID:	CAN_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	CAN_MODULE_ID	80 decimal (according to ref. [2])

* For the precise AUTOSAR Release 3.x and 4.x please see the release specific documentation.

3.1 Architecture Overview

The following figure shows where the CAN is located in the AUTOSAR architecture.

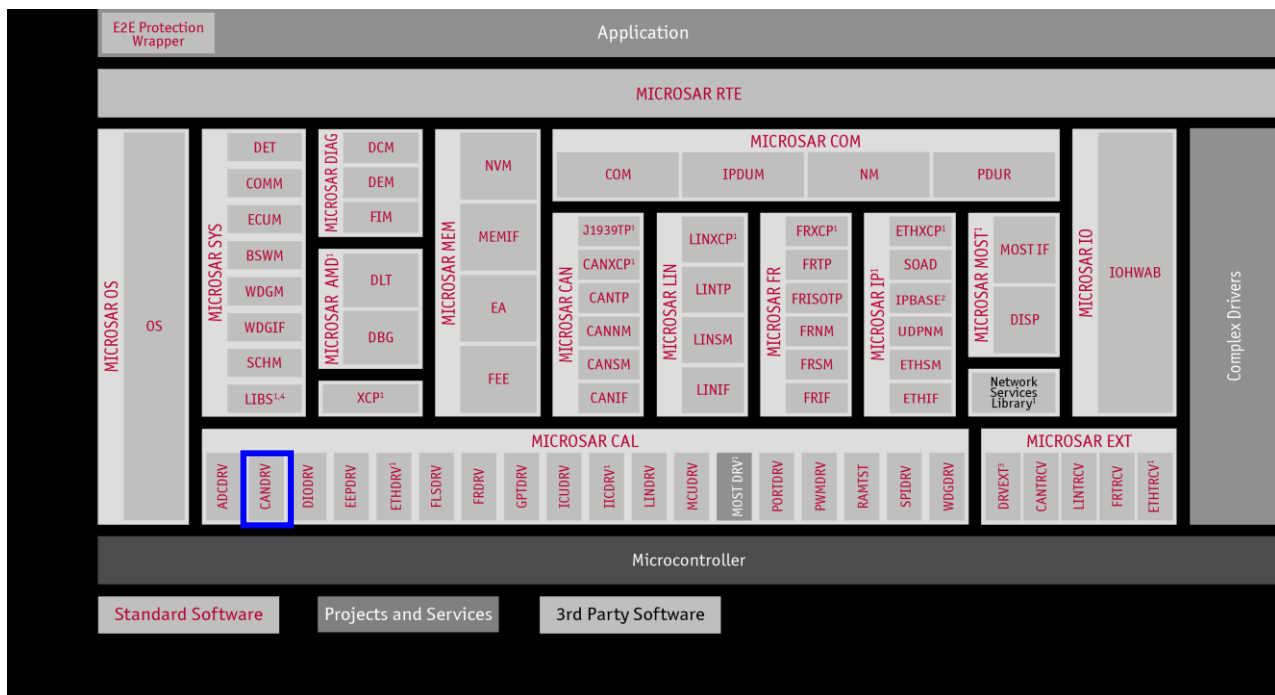


Figure 3-1 AUTOSAR architecture

The next figure shows the interfaces to adjacent modules of the CAN. These interfaces are described in chapter 6.

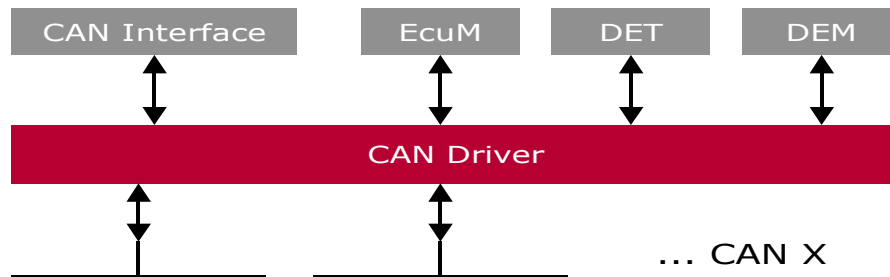


Figure 3-2 Interfaces to adjacent modules of the CAN

4. Functional Description

4.1 Features

The features listed in this chapter cover the complete functionality specified in [1].

The "supported" and "not supported" features are presented in the following table. For further information of not supported features also see chapter 8.

Feature Naming	Short Description	GENy	Generator GENy (extern Gen)	CFG5
Initialization				
Driver	General driver initialization function	■	■	-
Controller	Controller specific initialization function Can_InitController().	■	■	-
Communication				
Transmission	Transmitting CAN frames.	■	■	-
Transmit confirmation	Callback for successful Transmission.	■	■	-
Reception	Receiving CAN frames.	■	■	-
Receive indication	Callback for receiving Frame.	■	■	-
Controller Modes				
Sleep mode	Controller support sleep mode (power saving).	■	■	-
Wakeup over CAN	Controller support wakeup over CAN.	■	■	-
Stop mode	Controller support stop mode (passive to CAN bus).	■	■	-
Bus Off detection	Callback for Bus Off event.	■	■	-
Polling Modes				
Tx confirmation	Support polling mode for Transmit confirmation.	■	■	-
Reception	Support polling mode for Reception.	■	■	-
Wakeup	Support polling mode for Wakeup event.	■	■	-
Bus Off	Support polling mode for Bus Off event.	■	■	-
Mode	MICROSAR4x only: Support polling mode for mode transition.	■	■	-
Mailbox objects				
Tx BasicCAN	Standard mailbox to send CAN frames (Used by CAN Interface data queue).	■	■	-
Multiplexed Tx	Using 3 mailboxes for Tx BasicCAN mailbox (external priority inversion avoided).	□	□	-
Tx FullCAN	Separate mailbox for special Tx message used.	■	■	-

Maximum amount	Available amount of mailboxes (in case of no Rx FullCAN objects is used) This amount depends on the maximal number of mailbox objects of the hardware channel	Mailbox-object per channel: 32: Std ID: 29 Ext ID: 29 Mix ID: 27 Mailbox-object per channel: 64: Std ID: 61 Ext ID: 61 Mix ID: 59	Mailbox-object per channel: 32: Std ID: 31 Ext ID: 31 Mix ID: 29 Mailbox-object per channel: 64: Std ID: 63 Ext ID: 63 Mix ID: 61	-
Rx FullCAN	Separate mailbox for special Rx message used.	■	■	-
Maximum amount	Available amount of mailboxes (in case of no Rx FullCAN objects is used) This amount depends on the maximal number of mailbox objects of the hardware channel	Mailbox-object per channel: 32: Std ID: 29 Ext ID: 29 Mix ID: 27 Mailbox-object per channel: 64: Std ID: 61 Ext ID: 61 Mix ID: 59	Mailbox-object per channel: 32: Std ID: 31 Ext ID: 31 Mix ID: 29 Mailbox-object per channel: 64: Std ID: 63 Ext ID: 63 Mix ID: 61	-
Rx BasicCAN	Standard mailbox to receive CAN frames (depending on hardware, FIFO or shadow buffer supported).	■	■	-
Maximum amount	Available amount of BasicCAN objects.	1/2	1/2	-
Others				
DEM	Support Diagnostic Event Manager (error notification).	■	■	-
DET	Support Development Error Detection (error notification).	■	■	-
Version API	API to read out component version.	■	■	-
Maximum supported Controllers	Maximum amount of supported Controllers (hardware channels).	3	3	-

Cancellation of Tx objects	Support of Tx Cancellation (out of hardware). Avoid internal priority inversion.	<input type="checkbox"/>	<input type="checkbox"/>	-
Identical ID cancellation	Tx Cancellation also for identical IDs.	<input type="checkbox"/>	<input type="checkbox"/>	-
Standard ID types	Standard Identifier supported (Tx and Rx).	■	■	-
Extended ID types	Extended Identifier supported (Tx and Rx).	■	■	-
Mixed ID types	Standard and Extended Identifier supported (Tx and Rx).	■	■	-
Hardware Loop Check (Timeout monitoring)	To avoid possible endless loops (occur by hardware issue).	■	■	-
AutoSar extensions				
Individual Polling	Support individual polling mode (selectable for each mailbox separate).	<input type="checkbox"/> *	<input type="checkbox"/> *	-
Multiple Basic CAN	Support Multiple BasicCAN objects. This gives the possibility to use multiple Filters and optimize acceptance Filtering as well as avoid overrun.	<input type="checkbox"/> *	<input type="checkbox"/> *	-
Rx Queue	Support Rx Queue. This give the possibility to buffer received data in interrupt context but handle it asynchronous in polling task.	<input type="checkbox"/> *	<input type="checkbox"/> *	-
Secure Rx Buffer used	Special hardware buffer used to temporary save received data.	<input type="checkbox"/>	<input type="checkbox"/>	-
Hardware Loop Check by Application	“Hardware Loop Check” can be defined to be done by application (special API available)	■	■	-
Configurable “Nested CAN Interrupts”	Nested CAN interrupts allowed, and can be also switched to none-nested.	<input type="checkbox"/>	<input type="checkbox"/>	-
Report CAN_E_TIMEOUT DEM as DET	Report CAN_E_TIMEOUT (Hardware Loop Check / Timeout Monitoring) to DET instead of DEM.	■	■	-
Support Mixed ID	Force CAN driver to handle Mixed ID (standard and extended ID) at pre-compile-time to expand the ID type later on.	■	■	-
Optimize for one controller	Activate this for 1 controller systems when you never will expand to multi-controller. So that the CAN driver works more efficient	■	■	-
Dynamic FullCAN Tx ID (***)	Always write FullCAN Tx ID within CanWrite() API function. Deactivate this to optimize code when you do not use FullCAN Tx objects dynamically.	■	■	-

Size of Hw HandleType	Support 8bit or 16bit Hardware Handles depend on hardware usage.	■	■	-
Generic PreCopy	Support a callback function for receiving any CAN message (following callbacks could be suppressed)	■	■	-
Generic Confirmation	Support a callback function for successful transmission of any CAN message (following callbacks could be suppressed)	■	■	-
Get Hardware Status	Support a API to get hardware status Information (see Can_GetStatus())	■	■	-
Interrupt Category selection	Support Category 1 or Category 2 Interrupt Service Routines for OS	■	■	-
Common CAN	Support merge of 2 controllers in hardware to get more Rx FullCAN objects	□**	□**	-
Overrun Notification	Support DET or Application notification cause by overrun (overwrite) of an Rx message (BasicCAN and FullCAN)	■	■	-
RAM check	Support CAN mailbox RAM check	■	■	-
Multiple ECU configurations (***)	The feature Multiple ECU is usually used for nodes that exist more than once in a car. At power up the application decides which node should be realized.	■	■	-
Generic PreTransmit	Support a callback function with pointer to Data, right before this data will be written in Hardware mailbox buffer to send. (Use this to change data or cancel transmission)	■	■	-

Table 4-1 Supported features

■ Feature is supported

□ Feature is not supported

* HighEnd Licence only

** Project specific (may not be available)

*** Not supported for AutoSar version 4.0.3

4.2 Initialization

`Can_Init()` has to be called to initialize the CAN driver at power on and sets controller independent init values. This function has to be called before `Can_InitController()`.

MicroSar3 only: Use `Can_InitStruct()` to change the used baud rate and filter settings like given in the Initialization structure from the Tool. The used default set by `Can_InitMemory()` is the first structure. This API has to be called before `Can_InitController()` but after `Can_InitMemory()`.

MICROSAR401 only: baud rate settings given by `Can_InitController` parameter.

`Can_InitController()` initializes the controller, given as parameter, and can also be used to reinitialize. After this call the controller stays in stop-mode until the CAN Interface changes to start-mode.

`Can_InitMemory()` is an additional service function to reinitialize the memory to bring the driver back to a pre-power-on state (not initialized). Afterwards `Can_Init()` and `Can_InitController()` have to be called again. It is recommended to use this function before calling `Can_Init()` to secure that no startup-code specific pre-initialized variables affect the driver startup behavior.

4.3 Communication

`Can_Write()` is used to send a message over the mailbox object given as "Hth". The data, DLC and ID is copied into the hardware mailbox object and a send request is set. After sending the message the CAN Interface `CanIf_TxConfirmation()` function is called. Right before the data is copied in mailbox buffer the ID, DLC and data may be changed by `Appl_GenericPreTransmit()` callback.

When "Generic Confirmation" is activated the callback `Appl_GenericConfirmation()` will be called before `CanIf_TxConfirmation()` and the call to this can be suppressed by `Appl_GenericConfirmation()` return value.

For Tx messages the ID will be copied. (exception: feature "Dynamic FullCAN Tx ID" is deactivated, then the FullCAN Tx messages will be only set while initialization)

If the mailbox is currently sending the status busy will be returned. Then the message may be queued in the CAN interface (if feature is active).

If cancellation in hardware is supported the lowest priority ID inside currently sending object is canceled, and therefore re-queued in the CAN Interface.

`Appl_GenericPreCopy()` (if activated) is called and depend on return value also `CanIf_RxIndication()` as a CAN Interface callback, is called when a message is received. The receive information like ID, DLC and data are given as parameter.

When Rx Queue is activated the received messages (polling or interrupt context) will be queued (same queue over all channels). The Rx Queue will be read by calling `Can_Mainfunction_Read()` and the Rx Indication (like `CanIf_RxIndication()`) will be called out of this context. Rx Queue is used for Interrupt systems to keep Interrupt latency time short.

4.3.1 Mailbox Layout

The generation tool supports a flexible allocation of message buffers. In the following tables the possible mailbox layout is shown (the range for each mailbox types depend on the used mailboxes).

Object number	Object type	No. of Objects	Comment
1 – n	Tx Full CAN	0 - (MaxObj-3)	These objects are used to send a certain message. The user must define statically (Generation Tool) which CAN messages are located in such Tx FullCAN objects. The Generation Tool distributes the messages to the FullCAN objects.
n+1	Tx Basic CAN	1	These objects are used to send several messages. If the transmit message object is busy, the transmit request is stored in a CAN Interface queue (if activated)
n+2 - m	Unused	0 - (MaxObj-3)	These objects are not used. It depends on the configuration of receive and transmit objects if unused objects are available.
m+1 – o	Rx Full CAN	0 - (MaxObj-3)	These objects are used to receive specific CAN messages. The user defines statically (Generation Tool) that a CAN message should be received in a FullCAN message object. The Generation Tool distributes the message to the FullCAN objects.
o+1 – o+2	Rx Basic CAN	(0*)/2/4	All other CAN messages are received via the Basic CAN. For configurations with Standard Id or Extended Id the Basic CAN use 2 Message Objects, configurations with Mixed Id used 4 Message Objects.

Table 4-2 Hardware mailbox layout

The “CanObjectId” numbering is done in following order: FullCANTx, BasicTx, Unused, FullCANRx, BasicRx (like shown above). “CanObjectId’s” for next controller begin at end of last controller. Gaps in “CanObjectId” for unused mailboxes may occur. This objects must located at the position pictured in the table above.

It’s important to know that a configuration allways has to contain one BasicTx CAN per controller.

4.3.2 Acceptance Filter for BasicCAN

The Basic CAN consists for standard and extended Id of 2 message objects and 1 filter and for mixed Id of 4 message objects and 2 filters.

If no message should be received, select the “Multiple Basic CAN” feature and set the amount to 0. Otherwise the filter should be set to “close”. Use feature “Rx BasicCAN Support” to deactivate unused code (for optimization).

4.3.3 Remote Frames

Remote Frames rejected in Hardware	Remote Frames rejected in Software
x	

4.4 States / Modes

You can change the CAN cell mode via `Can_SetControllerMode()`. The last requested transition will be executed. The Upper layer has to take care about valid transitions.

Following modes changes are supported:

```
CAN_T_START
CAN_T_STOP
CAN_T_SLEEP
CAN_T_WAKEUP
```

4.4.1 Start Mode (Normal Running Mode)

This is the mode where communication is possible. This mode has to be set after Initialization because Controller is first in stop-mode.

4.4.2 Stop Mode

To enter stop mode the driver sets the init flag in the CAN control register to one. After this there is no CAN communication possible until stop mode is left.

4.4.3 Sleep Mode

There are two power-down modes available, the global power-down mode and the local power-down mode. The first is supported by all TMS570 DCAN derivatives, the second only if it is documented in the datasheet. This is because the CAN controller has not initially supported the local power-down mode. It was added to the DCAN cell since documented in the reference guide revision 0.30. More information about the selection of the different power down modes can found in table 7.1 "Power Down Mode Select".

To enter sleep mode the driver set the CAN control into power down mode. After this

`EcuM_CheckWakeup()` is called in case a wakeup over CAN event occurred.

`Can_Cbk_CheckWakeup()` can be called to check wakeup state.

`CAN_T_WAKEUP` has to be set with `Can_SetControllerMode()` to leave sleep mode.

4.4.4 Bus Off

`CanIf_ControllerBusOff()` is called when the controller detects a Bus Off event. The mode is automatically changed to stop mode. The upper layers have to care about returning to normal running mode by calling start mode.

4.5 Re-Initialization

A call to `Can_InitController()` cause a re-initialization of a dedicated CAN controller. Pending messages may be processed before the transition will be finished. A re-initialization is only possible out of Stop Mode and does not change to another Mode.

After re-initialization all CAN communication relevant registers are set to initial conditions.

4.6 CAN Interrupt Locking

`Can_DisableControllerInterrupts()` and `Can_EnableControllerInterrupts()` are used to disable and enable the controller specific Interrupt, Rx, Tx, Wakeup and Bus Off (/ Status) together. These functions can be called nested.

Be aware that this functions change the content of the CAN Interrupt inside the VIM (vector interrupt manager).

4.7 Main Functions

`Can_MainFunction_Write()`, `Can_MainFunction_Read()`, `Can_MainFunction_BusOff()` and `Can_MainFunction_Wakeup()` called by upper layer to poll the events if the specific polling mode is activated. Otherwise these functions return without action and the events will be handled in interrupt context.

When individual polling is activated only mailboxes that are configured as to be polled will be polled in the main functions "`Can_MainFunction_Write()`" and "`Can_MainFunction_Read()`" all others handled in interrupt context.

When Rx Queue is activated the queue is filled in interrupt or polling context like configured. But The processing (indications) will be done in "`Can_MainFunction_Read()`" context.

4.8 Error Handling

4.8.1 Development Error Reporting

Development errors are reported to DET using the service `Det_ReportError()`, if the pre-compile parameter `CAN_DEV_ERROR_DETECT == STD_ON`.

The tables below, shows the API ID and Error ID given as parameter for calling the DET.

Instance ID is always 0 because no multiple Instances are supported.

Errors reported to DET:	
Error ID	Short Description
<code>CAN_E_PARAM_POINTER</code>	API gets an illegal pointer as parameter.
<code>CAN_E_PARAM_HANDLE</code>	API get an illegal handle as parameter
<code>CAN_E_PARAM_DLC</code>	API get an illegal DLC as parameter
<code>CAN_E_PARAM_CONTROLLER</code>	API get an illegal controller as parameter
<code>CAN_E_UNINIT</code>	Driver API is used but not initialized
<code>CAN_E_TRANSITION</code>	Transition for mode change is illegal
<code>CAN_E_DATA_LOST</code> (value: 0x07, AutoSar extension)	Rx overrun (overwrite) detected
<code>CAN_E_PARAM_BAUDRATE</code> (value: 0x08, AutoSar extension)	Selected Baudrate is not valid

CAN_E_RXQUEUE (value: 0x10, AutoSar extension)	Rx Queue overrun (last received message is lost, and will not be received → increase queue size)
CAN_E_TIMEOUT_DET (value: 0x11, AutoSar extension)	Same as CAN_E_TIMEOUT for DEM but this is notified to DET due to switch "CAN_DEV_TIMEOUT_DETECT" is set to STD_ON (see configuration options)

Table 4-3 Errors reported to DET

API from which the errors reported to DET:	
API ID	Functions using that ID
CAN_VERSION_ID	Can_GetVersionInfo()
CAN_INIT_ID	Can_Init()
CAN_INITCTR_ID	Can_InitController()
CAN_SETCTR_ID	Can_SetControllerMode()
CAN_DIINT_ID	Can_DisableControllerInterrupts()
CAN_ENINT_ID	Can_EnableControllerInterrupts()
CAN_WRITE_ID	Can_Write(), Can_CancelTx()
CAN_TXCNF_ID	CanHL_TxConfirmation()
CAN_RXINDI_ID	CanBasicCanMsgReceived(), CanFullCanMsgReceived()
CAN_CTRBUSOFF_ID	CanHL_ErrorHandling()
CAN_CKWAKEUP_ID	CanHL_WakeUpHandling(), Can_Cbk_CheckWakeup()
CAN_MAINFCT_WRITE_ID	Can_MainFunction_Write()
CAN_MAINFCT_READ_ID	Can_MainFunction_Read()
CAN_MAINFCT_BO_ID	Can_MainFunction_BusOff()
CAN_MAINFCT_WU_ID	Can_MainFunction_Wakeup()
CAN_MAINFCT_MODE_ID	Can_MainFunction_Mode()
CAN_CHANGE_BR_ID	Can_ChangeBaudrate()
CAN_CHECK_BR_ID	Can_CheckBaudrate()
CAN_HW_ACCESS_ID (value: 0x20, AUTOSAR extension)	Used when hardware is accessed (call context is unknown)

Table 4-4 API from which the Errors are reported

4.8.1.1 Parameter Checking

AUTOSAR requires that API functions check the validity of their parameters (Refer to [1]). These checks are for development error reporting and can be enabled and disabled separately. Refer to the configuration chapter where the enabling/disabling of the checks is described. Enabling/disabling of single checks is an addition to the AUTOSAR standard which requires enable/disable the complete parameter checking via the parameter CAN_DEV_ERROR_DETECT.

4.8.1.2 Overrun/Overwrite Notification

As AUTOSAR extension the overrun detection may be activated by configuration tool. The notification can be configured to call an DET (MICROSAR4x) or Application call *ApplCanOverrun()* or *ApplCanFullCanOverrun()*.



Info

The overrun/overwrite behavior of FullCAN and BasicCAN mailboxes differs from each other.

For **FullCAN** mailboxes: the old message is overwritten by a new incoming one. The old message gets lost.

For **BasicCAN** mailboxes: the old message is not overwritten by a new incoming one. The new message gets lost.

4.8.2 Production Code Error Reporting

Production code related errors are reported to DEM using the service `Dem_ReportErrorStatus()`, if the pre-compile parameter `CAN_PROD_ERROR_DETECT == STD_ON`.

The table below shows the API ID and Error ID given as parameter for calling the DEM. This means that the Error IDs listed below could occur in most of the API IDs.

Event ID	Event Status	Short Description
CAN_E_TIMEOUT	DEM_EVENT_STATUS_FAILED	Timeout in "Hardware Loop Check" occurred, hardware has to be checked or timeout is too short.

Table 4-5 Errors reported to DEM

4.8.2.1 Hardware Loop Check / Timeout Monitoring

The feature "Hardware Loop Check" is used to break endless loops caused by hardware issue. This feature is configurable see Chapter 7 and also Timeout Duration description.

The Hardware Loop Check will be handled by CAN driver internal except when setting "Hardware Loop Check by Application" is activated.

Loop Name / source	Short Description
kCanLoopIrqReq	Loop over all pending interrupts (Rx, Tx, BusOff). This loop is used to stay in interrupt context until all interrupt events are executed. This saves time because no reentering of ISR is needed to execute more than one event.
kCanLoopBusyReq	Loop used to indicate can cell busy state. Maximum duration of the loop should be at most 3-6 CAN CLK periods.

kCanLoopSleep	<p>MICROSAR3:</p> <ul style="list-style-type: none"> - Used while transition in mode 'SLEEP'. - Call context: Can_SetControllerMode() - Expected duration: after the request to enter sleep mode is set, the CAN cell finishes all transmission and reception processes. After this it waits until a bus idle state is recognized. Then the Init-bit will be set to 1 to indicate that the sleep mode has been entered. <p>The expected duration lasts one CAN frame including interframe space.</p> <p>MICROSAR4:</p> <p>Used for short time mode transition blocking (short synchronous timeout). Same value for kCanLoopSleep and kCanLoopWakeup. No Issue when timeout occur.</p>
kCanLoopWakeup	<p>MICROSAR3:</p> <ul style="list-style-type: none"> - Used while transition in mode 'WAKEUP'. - Call context: Can_SetControllerMode() - Expected duration: If the CAN cell leaves sleep mode, the Init-Bit will be cleared. After this the DCAN module waits until it detects 11 consecutive recessive bits on the CAN_RX pin and then goes Bus-Active again. <p>MICROSAR4:</p> <p>Used for short time mode transition blocking (short synchronous timeout). Same value for kCanLoopSleep and kCanLoopWakeup. No Issue when timeout occur.</p>

Table 4-6 Hardware Loop Check

In the generation tool the user has to set a value for the number of loops until that the time-out duration occurs. This must be the value of the most often covered hardware loop. It is the "loop over all pending interrupts" (loop name kCanLoopIrqReq) and has the size of the maximum number of message objects. For the TMS570 it is 64 and with a security buffer a good value to use will be 100.

4.8.3 CAN RAM Check

The CAN driver supports a check of the CAN controller's mailboxes. The CAN controller RAM check is called internally every time a power on is executed within function Can_InitController(), or a Bus-Wakeup event happen. The CAN driver verifies that no used mailboxes are corrupt. A mailbox is considered corrupt if a predefined pattern is written to the appropriate mailbox registers and the read operation does not return the expected pattern. If a corrupt mailbox is found the function Appl_CanCorruptMailbox() is called. This function tells the application which mailbox is corrupt.

After the check of all mailboxes the CAN driver calls the call back function Appl_CanRamCheckFailed() if at least one corrupt mailbox was found. The application must decide if the CAN driver disables communication or not by means of the call back function's return value. If the application has decided to disable the communication there is no possibility to enable the communication again until the next call to Can_Init().

The CAN RAM check functionality itself can be activated via Generation Tool.

4.8.4 Hardware Specific

There are some restrictions in the hardware as not all the CAN driver features are supported. The not supported features are:

1. Cancel in Hardware:

The hardware does not provide support to request a cancelation for an already requested (pending) transition. Consequently it is not possible to support cancel in hardware.

2. Multiplexed Transmitting:

The mailboxes are prioritized by hardware index and not by CAN ID. Consequently it is not possible to support multiplexed transmission.

3. Supported Controllermode:

The CAN driver was designed to run in privileged mode only. There is no support for user mode.

4. Errata DCAN#22:

There could happen that an incorrect payload (data bytes) is stored in mailbox under certain conditions. This is a HW issue present in several revisions (A and earlier). For details please refer to silicon errata. The CAN driver implements the first workaround proposal no.1 as it can be applied also to the families without local power down-mode and it does not disturb the other peripherals.

For that purpose the user has to calibrate a “6 NOPs” dummy loop, i.e. the software has to wait for at least 6 CAN clocks cycles (corresponding to the CAN clock input and not CAN bus). The 6 NOPs are not optimized and the group cannot take less than 6 CPU cycles even if the core has a pipeline.

The number of iterations through the “6 NOPs” has the following formula:

$$\text{ErrataDcan22Iterations} = \text{CPU_CLOCK} / \text{CAN_CLOCK}$$

Afterwards the calculated value corresponding to ErrataDcan22Iterations has to be entered in the configuration (see 7.4.2).



Info

If the used version of the silicon is not affected by this issue, the workaround can be disabled as followings:

1) A user config file has to be created; this will be installed in the CAN driver component of the configuration (see 7.4.2).

2) This used config file will contain the following line:
`#define C_DISABLE_DCAN_ISSUE22_WORKAROUND.`

5. Integration

This chapter gives necessary information for the integration of the MICROSAR CAN into an application environment of an ECU.

5.1 Scope of Delivery

The delivery of the CAN contains the files, which are described in the chapter's 5.1.1 and 5.1.2:

Dependent on library or source code delivery the marked (+) files may not be delivered.

5.1.1 Static Files

File Name	Description
(+) canproto.h	This is an internal header file which should not be included outside this module
(+) Can.c	This is the source file of the CAN. It contains the implementation of CAN module functionality.
(+) Can.lib	This is the library build out of Can.c, Can.h and CanProto.h
Can.h	This is the header file of the CAN module (include API declaration)
Can_Hooks.h	This is the header file to define the Hook-functions or macros.
Can_Irq.c	This is the interrupt declaration and callout file (supports interrupt configuration as link time settings)

Table 5-1 Static files

5.1.2 Dynamic Files

The dynamic files are generated by the configuration tool [GENy].

File Name	Description
Can_Cfg.h	Generated header file, contains some type, prototype and pre-compile settings
Can_Lcfg.c	Generated file contains link time settings.
Can_PBcfg.c	Generated file contains post build settings.
Can_DrvGeneralTypes.h	Generated file contains CAN driver part of Can_GeneralTypes.h (supported by Integrator)

Table 5-2 Generated files

5.2 Include Structure

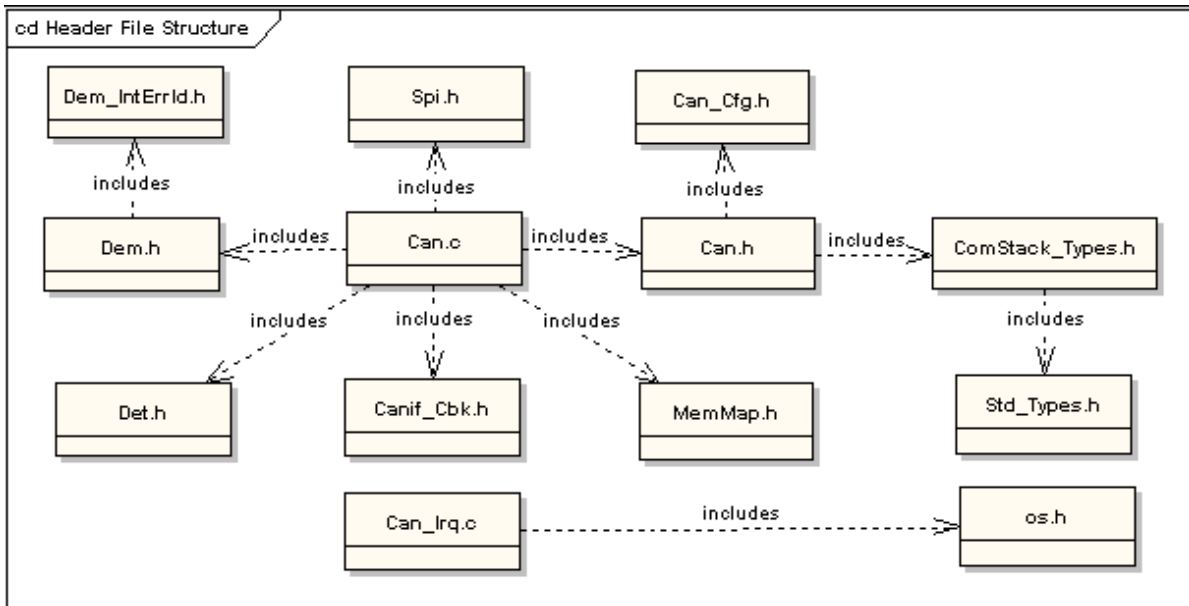


Figure 5-1 Include Structure (AUTOSAR)

Deviation from AUTOSAR specification:

- > Additionally the EcuM_Cbk.h is included by Can_Cfg.h (needed for wakeup notification API).
- > ComStack_Types.h included by Can_Cfg.h, because the specified types have to be known in generated data as well.
- > MICROSAR4x only: Os.h will be included by Can_Cfg.h because of used data-types
- > Spi.h is not yet used.
- > Additionally the file Can_Hooks.h will be included by Can.h.
- > MICROSAR403 only: Can_GeneralTypes.h will be included by Can_Cfg.h not by Can.h direct.

5.3 Critical Sections

The AUTOSAR standard provides with the BSW Scheduler a BSW module, which handles entering and leaving critical sections.

For more information about the BSW Scheduler please refer to [3]. When the BSW Scheduler is used the CAN Driver provides critical section codes that have to be mapped by the BSW Scheduler to following mechanism:

Critical Section Define	Description
CAN_EXCLUSIVE_AREA_0	<p><u>CanSingleGlobalInterruptDisable/Restore():</u> Using inside Can_MainFunction_Write() to secure a consistent polling of a transmit object.</p> <ul style="list-style-type: none"> ■ Duration is short (< 10 instructions). ■ No API call of other BSW inside. <p><u>CanNestedGlobalInterruptDisable/Restore():</u> Using inside CanHL_TxConfirmation to secure a consistent release of a transmit object after successful transmission.</p> <ul style="list-style-type: none"> ■ Duration is short (< 10 instructions) ■ No API call of other BSW inside.
CAN_EXCLUSIVE_AREA_1	<p>Using inside Can_DisableControllerInterrupts() and Can_EnableControllerInterrupts() to secure Interrupt counters for nested calls.</p> <ul style="list-style-type: none"> ■ Duration is short (< 10 instructions) ■ No API call of other BSW inside. ■ Disable global interrupts – or – Empty in case Can_Disable/EnableControllerInterrupts() are called within context with lower or equal priority than CAN interrupt.
CAN_EXCLUSIVE_AREA_2	<p>Using inside Can_Write() to secure software states of transmit objects.</p> <ul style="list-style-type: none"> ■ Only when no Vector CAN Interface is used. ■ Duration is medium (< 60 instructions) ■ No API call of other BSW inside. ■ Disable global interrupts - or - Disable CAN interrupts and does not call function reentrant.
CAN_EXCLUSIVE_AREA_3	<p>Using inside Tx confirmation to secure state of transmit object in case of cancellation. (Only used when Vector Interface Version smaller 4.10 used)</p> <ul style="list-style-type: none"> ■ Duration is short (< 20 instructions) ■ Call to CanIf_CancelTxConfirmation() inside (no more calls in CanIf). ■ Disable global interrupts - or - Disable CAN interrupts and do not call function Can_Write() within.
CAN_EXCLUSIVE_AREA_4	<p>Using inside received data handling (Rx Queue treatment) to secure Rx Queue counter and data.</p> <ul style="list-style-type: none"> ■ Duration is short (< 20 instructions) ■ No API call of other BSW inside. ■ Disable Global Interrupts - or - Disable all CAN interrupts.

CAN_EXCLUSIVE_AREA_5	Using inside wakeup handling to secure state transition. (Only in wakeup polling mode) <ul style="list-style-type: none"> ■ Duration is short Duration is short (< 10 instructions). ■ Call to DET inside. ■ Disable global interrupts (do no use here CAN interrupt locks)
CAN_EXCLUSIVE_AREA_6	Using inside Can_SetControllerMode() and BusOff to secure state transition. <ul style="list-style-type: none"> ■ Duration is medium (< 100 instructions) ■ No API call of other BSW inside. ■ Use CAN interrupt locks here, when the API for one controller is not called in a context higher than the CAN interrupt or Disable global interrupts

Table 5-3 Critical Section Codes

5.4 Compiler Abstraction and Memory Mapping

The objects (e.g. variables, functions, constants) are declared by compiler independent definitions – the compiler abstraction definitions. Each compiler abstraction definition is assigned to a memory section.

The following table contains the memory section names and the compiler abstraction definitions defined for the CAN Interface and illustrates their assignment among each other.

Memory Mapping Sections	CAN_CODE	CAN_STATIC_CODE	CAN_CONST	CAN_CONST_PBCFG	CAN_VAR_NOINIT	CAN_VAR_INIT	CAN_INT_CTRL	CAN_REG_CANCEL	CAN_RX_TX_DATA	CAN_APPL_CODE	CAN_APPL_CONST	CAN_APPL_VAR
CAN_START_SEC_CODE	■											
CAN_STOP_SEC_CODE												
CAN_START_SEC_STATIC_CODE		■										
CAN_STOP_SEC_STATIC_CODE												
CAN_START_SEC_CONST_8BIT			■									
CAN_STOP_SEC_CONST_8BIT												
CAN_START_SEC_CONST_16BIT			■									
CAN_STOP_SEC_CONST_16BIT												
CAN_START_SEC_CONST_32BIT			■									
CAN_STOP_SEC_CONST_32BIT												

CAN_START_SEC_CONST_UNSPECIFIED			■									
CAN_STOP_SEC_CONST_UNSPECIFIED												
CAN_START_SEC_PBCFG				■								
CAN_STOP_SEC_PBCFG												
CAN_START_SEC_PBCFG_ROOT				■								
CAN_STOP_SEC_PBCFG_ROOT												
CAN_START_SEC_VAR_NOINIT_UNSPECIFIED					■							
CAN_STOP_SEC_VAR_NOINIT_UNSPECIFIED												
CAN_START_SEC_VAR_INIT_UNSPECIFIED						■						
CAN_STOP_SEC_VAR_INIT_UNSPECIFIED												
CAN_START_SEC_CODE_APPL										■		
CAN_STOP_SEC_CODE_APPL												

Table 5-4 Compiler abstraction and memory mapping

The Compiler Abstraction Definitions `CAN_APPL_CODE`, `CAN_APPL_VAR` and `CAN_APPL_CONST` are used to address code, variables and constants which are declared by other modules and used by the CAN driver.

These definitions are not mapped by the CAN driver but by the memory mapping realized in the CAN Interface or direct by application.

`CAN_CODE`: used for CAN module code.

`CAN_STATIC_CODE`: used for CAN module local code.

`CAN_CONST`: used for CAN module constants.

`CAN_CONST_PBCFG`: used for CAN module constants in Post-Build section.

`CAN_VAR_*`: used for CAN module variables.

`CAN_INT_CTRL`: is used to access the CAN interrupt controls.

`CAN_REG_CANCELL`: is used to access the CAN cell itself.

`CAN_RX_TX_DATA`: access to CAN Data buffers located in RAM.

`CAN_APPL_*`: access to higher layers.

5.5 Hardware Specific Hints

1.1.1 Initialisation of the VIM-register

The register of the Vectored Interrupt Manager (VIM) will be modified by the CAN-driver to enable or disable the CAN-Interrupts. The initial value of the register has to be set by the application. If the application does not enable the CAN-Interrupts at startup, the interrupts stays disabled the whole time.

1.1.2 Hardware specific topics

For details regarding the hardware specific topics please check the appropriate hardware manual of concrete derivative.

6. API Description

6.1 Interrupt Service Routines provided by CAN

Depend on the settings in Tools component Hw_Tms470Cpu, the interrupt routine is given by the driver or by Operating System (selection see below).

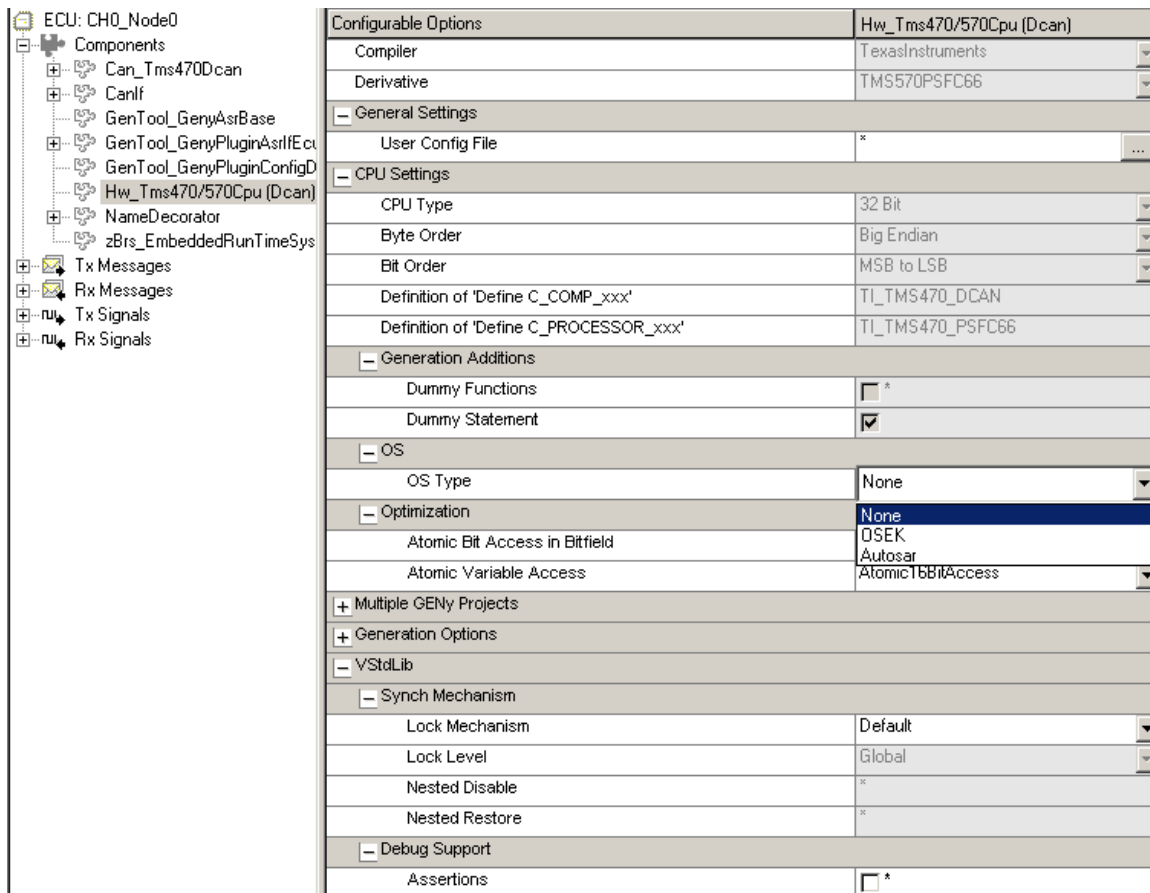


Figure 6-1 Select OS Type

There is the possibility to choose OS Type. Please select "None" for using no OS, "Autosar" for AUTOSAR OS or "OSEK" for OSEK OS systems.

6.1.1 OSEK (OS)

This means to include osek.h.

Switch: V_OSTYPE_OSEK

6.1.2 AutoSar (OS)

Os.h header file is used.

Switch: V_OSTYPE_AUTOSAR

6.1.3 None (OS)

Choose “None” for OS Type, to include no Os header files and have no category 2 interrupt.

Switch: V_OSTYPE_NONE

6.1.4 Type of Interrupt Function

See Chapter “Component Settings” for configuration aspects.

- Category 2 (only for OSEK OS or AUTOSAR OS):
A macro “ISR(‘CanIsr_x)’” will be used to declare ISR function call. The name given as parameter for interrupt naming (x = Physical CAN Channel number). For macro definition see OS specification. The OS has full control of the ISR.
switch: C_ENABLE_OSEK_OS_INTCAT2
- Category 1: Using OS with category 1 interrupts need an Interface layer handling these interrupts in task context like defined in BSW00326 (AUTOSAR_SRS_General).
The ISR is declared as void ISR(void) and has to be called by interrupt controller in case of an CAN interrupt.
switch: C_DISABLE_OSEK_OS_INTCAT2
- Void-Void Interrupt Function:
Like in Category 1 the Interrupt is not handled by OS. With this CAN-driver there is no difference to Category 1: the ISR is declared as void ISR(void) and has to be called by interrupt controller in case of an CAN interrupt.

6.1.5 CanIsr_<CH>

Prototype	
void CanIsr_<CH>(void)	
Parameter	
---	---
Return code	
---	---
Functional Description	
This function has to be called in scope of interrupt processing. It handles the Tx, Rx and BusOff interrupts and clears the pending flags of the CAN cell and pending bits of the interrupt controller.	
Particularities and Limitations	
<div style="display: flex; align-items: flex-start;"> <div style="width: 1em; margin-right: 5px;">■</div> <div>This function can be used directly as interrupt routine in an interrupt vector table for category 1 interrupts.</div> </div>	

6.2 Services provided by CAN

The CAN API consists of services, which are realized by function calls.

6.2.1 Can_GetVersionInfo

Prototype	
void Can_GetVersionInfo(Std_VersionInfoType *versioninfo);	
Parameter	
versioninfo	pointer to the structure including the version information <pre>typedef struct { uint16 vendorID; uint16 moduleID; MicroSar3 only: uint8 instanceID; uint8 sw_major_version; /*BCD code*/ uint8 sw_minor_version; /*BCD code*/ uint8 sw_patch_version; /*BCD code*/ } Std_VersionInfoType;</pre>
Return code	
-	-
Functional Description	
Function to get version information	
Particularities and Limitations	
> - The function is only available if enabled at build time (CAN_VERSION_INFO_API = ON)	

6.2.2 Can_Init

Prototype	
void Can_Init(const Can_ConfigType *Config)	
Parameter	
Config	Pointer to the structure including configuration data. In case of Multiple ECU configuration feature is used, for each Identity one "Config" structure exists and has to be chosen here
Return code	
-	-
Functional Description	
This function initializes global CAN driver variables during ECU start-up.	

Particularities and Limitations

- > Has to be called during start-up before CAN communication.
- > Must be called before calling Can_InitController()
- > Multiple ECU configuration pointer for "Config" does only work with none Post-Build variants
- > Can_InitMemory() has to be called before.

6.2.3 Can_InitController

Prototype

MICROSAR3:

```
void Can_InitController( uint8 Controller, const Can_ControllerConfigType*
Config )
```

MICROSAR401:

```
void Can_InitController( uint8 Controller, const
Can_ControllerBaudrateConfigPtrType Config )
```

Parameter

Controller	The controller to be initialized
Config	MICROSAR3: Pointer to the structure including configuration data. MICROSAR401: Pointer to baudrate structure.

Return code

-	-
---	---

Functional Description

This function initializes controller specific CAN driver registers and variables and leave in the stop mode with a fully initialized CAN controller ready to change in running mode.

Particularities and Limitations

- > Has to be called during start-up before CAN communication, but after Can_Init().
- > Must not be called within sleep mode

6.2.4 Can_SetControllerMode

Prototype

```
Can_ReturnType Can_SetControllerMode( uint8 Controller, Can_StateTransitionType
Transition )
```

Parameter

Controller	The controller to change mode
Transition	Mode transition request

Return code	
state	Result of changing mode can be: CAN_OK, CAN_NOT_OK
Functional Description	
This function switch to one of the following modes: Sleep, Stop, Running.	
Particularities and Limitations	
<ul style="list-style-type: none"> > Called by higher layer CAN interface > Must not be called within CAN driver context like Rx, Tx or Bus Off callouts. 	

6.2.5 Can_Write

Prototype	
Can_ReturnType Can_Write (uint8 Hth, const Can_PduType *PduInfo);	
Parameter	
Hth	Handle of the mailbox to send message
PduInfo	Information about outgoing message (ID, DLC, data)
Return code	
-state	- Result of transmission request can be: CAN_OK, CAN_NOT_OK, CAN_BUSY
Functional Description	
Called by the interface layer to send a message over CAN.	
Particularities and Limitations	
<ul style="list-style-type: none"> > Must not be called nested > Called with disabled interrupts (for data security CAN Interface have to do this and therefore not needed in driver twice) 	

6.2.6 Can_DisableControllerInterrupts

Prototype	
void Can_DisableControllerInterrupts(uint8 Controller);	
Parameter	
Controller	The controller to disable interrupts for
Return code	
-	-
Functional Description	
Service function to disable interrupts (e.g. to secure data)	
Particularities and Limitations	

6.2.7 Can_EnableControllerInterrupts

Prototype	
void Can_EnableControllerInterrupts(uint8 Controller);	
Parameter	
Controller	The controller to re enable interrupts for
Return code	
-	-
Functional Description	
Service function to re enable interrupts (e.g. to secure data)	
Particularities and Limitations	

6.2.8 Can_MainFunction_Write

Prototype	
void Can_MainFunction_Write(void);	
Parameter	
-	-
Return code	
-	-
Functional Description	
Service function to poll Tx confirmation (for all controller and Tx mailboxes). Has to be called by Higher Layer. Notify CAN Interface about Tx confirmations.	
Particularities and Limitations	
> Must not be called nested and only out of lower or equal task level as Can_Write().	

6.2.9 Can_MainFunction_Read

Prototype	
void Can_MainFunction_Read(void);	
Parameter	
-	-
Return code	
-	-

Functional Description	
Service function to poll reception (for all controller and Rx mailboxes). Also use to read out Rx Queue messages queued within interrupt context.	
Has to be called by Higher Layer.	
Notify CAN Interface about Rx indications.	
Particularities and Limitations	
> Must not be called nested.	

6.2.10 Can_MainFunction_BusOff

Prototype	
void Can_MainFunction_BusOff(void);	
Parameter	
-	-
Return code	
-	-
Functional Description	
Service function to poll Bus Off event (over all controller).	
Has to be called by Higher Layer.	
Calls "CanIf_ControllerBusOff" in case of Bus Off occurred.	
Particularities and Limitations	
> Must not be called nested.	

6.2.11 Can_MainFunction_Wakeup

Prototype	
void Can_MainFunction_Wakeup(void);	
Parameter	
-	-
Return code	
-	-
Functional Description	
Service function to poll Wakeup event (over all controller).	
Has to be called by Higher Layer.	
Calls "CanIf_SetWakeupEvent" in case of wakeup event occur.	
Particularities and Limitations	
> Must not be called nested.	

6.2.12 Can_MainFunction_Mode (MICROSAR4x only)

Prototype	
<code>void Can_MainFunction_Mode(void);</code>	
Parameter	
-	-
Return code	
-	-
Functional Description	
Service function to poll Mode changes (over all controller – asynchronous when not done in Can_SetControllerMode()).	
Polling of Mode change events notified to upper layer.	
Particularities and Limitations	
> Must not be called nested.	

6.2.13 Can_ChangeBaudrate (MICROSAR403 only)

Prototype	
<code>Std_ReturnType Can_ChangeBaudrate(uint8 Controller, const uint16 Baudrate);</code>	
Parameter	
Controller	The controller to change Baudrate
Baudrate	Baudrate to be set
Return code	
Std_ReturnType	E_NOT_OK: Baudrate is not set. E_OK: Baudrate is set.
Functional Description	
This service shall change the baudrate of the CAN controller. And reinitialize the CAN controller.	
Particularities and Limitations	
> Must not be called nested.	

6.2.14 Can_CheckBaudrate (MICROSAR403 only)

Prototype	
<code>Std_ReturnType Can_CheckBaudrate(uint8 Controller, const uint16 Baudrate);</code>	
Parameter	
Controller	The controller to check Baudrate
Baudrate	Baudrate to be checked
Return code	
Std_ReturnType	E_NOT_OK: Baudrate is not available. E_OK: Baudrate is available.

Functional Description	
This service shall check if a certain baudrate of the CAN controller is supported.	
Particularities and Limitations	
<ul style="list-style-type: none"> > Must not be called nested. > Only available when "CanChangeBaudrateApi" is activated. 	

6.2.15 Can_InitMemory (None AUTOSAR API)

Prototype	
<code>void Can_InitMemory(void);</code>	
Parameter	
-	-
Return code	
-	-
Functional Description	
Service function to initialize module global variables for power up situation. Call this functionality to simulate a power up behavior. (Afterwards call normal initialization functions)	
Particularities and Limitations	

6.2.16 Can_InitStruct (None AUTOSAR API)

Prototype	
<code>void Can_InitStruct(uint8 Controller, uint8 Index);</code>	
Parameter	
Controller	Controller to change Initialization structure
Index	Index of the Initialization structure to be used
Return code	
-	-
Functional Description	
Service function to change the Initialization structure setup by Tool. It contains information about the baud rate and filter settings. It is necessary to call Can_InitController afterwards to activate these settings.	
Particularities and Limitations	
<ul style="list-style-type: none"> > Call this function in between Can_Init () and Can_InitController(). > MicroSar3 only 	

6.2.17 Can_Cbk_CheckWakeup, Can_CheckWakeup

Prototype	
MICROSAR3: <code>Std_ReturnType Can_Cbk_CheckWakeup(uint8 Controller);</code> MICROSAR4x: <code>Std_ReturnType Can_CheckWakeup(uint8 Controller);</code>	
Parameter	
Controller	Controller to check for wakeup
Return code	
Std_ReturnType	E_OK when wakeup is cause by Controller or E_NOT_OK
Functional Description	
Service function to check occurrence of wakeup event on this controller.	
Particularities and Limitations	

6.2.18 ApplCanTimerStart (None AUTOSAR API)

Prototype	
<code>void ApplCanTimerStart (CanChannelHandle controller, uint8 source);</code> <code>void ApplCanTimerStart (uint8 source);</code> when using “Optimize for one controller”	
Parameter	
controller	controller on which the hardware observation occur
source	source for the hardware observation (see Hardware Loop Check / Timeout Monitoring)
Return code	
void	
Functional Description	
Service function to start an observation timer (see Chapter Hardware Loop Check).	
Particularities and Limitations	
Only available when “Hardware Loop Check by Application” is activated.	

6.2.19 ApplCanTimerLoop (None AUTOSAR API)

Prototype	
<code>Can_ReturnType ApplCanTimerLoop (CanChannelHandle controller, uint8 source);</code> <code>Can_ReturnType ApplCanTimerLoop (uint8 source);</code> when using “Optimize for one controller”	

Parameter	
controller	controller on which the hardware observation occur
source	source for the hardware observation (see Hardware Loop Check / Timeout Monitoring)
Return code	
Can_ReturnType (uint8)	E_OK when loop shall be continue E_NOT_OK when loop shall break to continue (E_NOT_OK should only be used when timeout occur due to hardware issue. Afterwards special action is needed see Chapter Hardware Loop Check)
Functional Description	
Service function to check timeout for the hardware observation (see Chapter Hardware Loop Check).	
Particularities and Limitations	
Only available when “Hardware Loop Check by Application” is activated.	

6.2.20 ApplCanTimerEnd (None AUTOSAR API)

Prototype	
<pre>void ApplCanTimerEnd (CanChannelHandle controller, uint8 source); void ApplCanTimerEnd (uint8 source); when using “Optimize for one controller”</pre>	
Parameter	
controller	controller on which the hardware observation occur
source	source for the hardware observation (see Hardware Loop Check / Timeout Monitoring)
Return code	
void	
Functional Description	
Service function to end an observation timer (see Chapter Hardware Loop Check).	
Particularities and Limitations	
Only available when “Hardware Loop Check by Application” is activated.	

6.2.21 Appl_GenericPrecopy (None AUTOSAR API)

Prototype	
<pre>Can_ReturnType Appl_GenericPrecopy (uint8 controller, Can_IdType ID, uint8 DLC, const uint8 *pData);</pre>	
Parameter	
Controller	Controller receive the data
ID	ID of the received message. In case of extended ID and mixed ID systems the highest bit (bit 32) is used to mark an extended ID (bit 32 is set).
DLC	DLC of the received message
pData	pData pointer to the data of the received message

Return code	
Can_ReturnType	CAN_OK when indication of the message should be called afterwards (notification to higher layer), or CAN_NOT_OK to break the Rx path.
Functional Description	
Application function inform about incoming Rx messages with complete data set	
Particularities and Limitations	
<ul style="list-style-type: none"> > This API is only usable when switch “#define CAN_GENERIC_PRECOPY STD_ON” is set by Generation tool. > pData is read only and must not be accessed with write operations. 	

6.2.22 Appl_GenericConfirmation (None AUTOSAR API)

Prototype	
Can_ReturnType Appl_GenericConfirmation(PduIdType PduId);	
Parameter	
PduId	PduId : Handle of the PDU (specify the message).
Return code	
Can_ReturnType	CAN_NOT_OK: No further CanInterface Confirmation will be called. CAN_OK: CanInterface Confirmation will be called.
Functional Description	
Application function inform about Tx messages is send out.	
Particularities and Limitations	
<ul style="list-style-type: none"> > This API is only usable when switch “#define CAN_GENERIC_CONFIRMATION STD_ON” is set by Generation tool. > If GenericConfirmation and TransmitBuffer (set in CanIf) is used the switch CAN_CANEL_SUPPORT_API is also needed (set in CanIf) otherwise a compiler error occur. > PduId is read only and must not be accessed with write operations. 	

6.2.23 Appl_GenericPreTransmit (None AUTOSAR API)

Prototype	
void Appl_GenericPreTransmit(uint8 Controller, Can_PduInfoPtrType_var DataPtr);	
Parameter	
Controller	controller on which the hardware observation occur
DataPtr	pointer to a Can_PduType structure include ID, DLC, Pdu, data-pointer
Return code	
void	

Functional Description
Application function to change send data (e.g. add CRC to data)
Particularities and Limitations
<p>> This API is only usable when switch “#define CAN_GENERIC_PRETRANSMIT STD_ON” is set by Generation tool.</p>

6.2.24 Can_GetStatus (None AUTOSAR API)

Prototype	
Uint8 Can_GetStatus(uint8 controller);	
Parameter	
Controller	Controller for status information
Return code	
uint8 status	CAN_STATUS_SLEEP CAN_STATUS_STOP CAN_STATUS_INIT CAN_STATUS_INCONSISTENT CAN_STATUS_WARNING (need C_ENABLE_EXTENDED_STATUS) CAN_STATUS_PASSIVE CAN_STATUS_BUSOFF
Functional Description	
<p>return the status of the hardware.</p> <p>Only one of the status-bits Sleep, Stop, Bus Off, Passive, Warning is set. Init bit is always set if controller is initialized.</p> <p>Sleep has the highest priority, error warning the lowest.</p> <p>CAN_STATUS_INCONSISTENT will be set if one Common CAN channel is not stop or sleep.</p> <p>“status” can be analyzed by us of macro API:</p> <p>CanHwlsOk(status): return “true” in case no warning, passive or bus off occurred.</p> <p>CanHwlsWarning(status): return “true” in case of waning status.</p> <p>CanHwlsPassive(status): return “true” in case of passive status.</p> <p>CanHwlsBusOff(status): return “true” in case of bus off status (may be already false in Notification).</p> <p>CanHwlsWakeup(status): return “true” in case of not in sleep mode.</p> <p>CanHwlsSleep(status): return “true” in case of sleep mode.</p> <p>CanHwlsStop(status): return “true” in case of stop mode.</p> <p>CanHwlsStart(status): return “true” in case of not in stop mode.</p> <p>CanHwlsInconsistent(status): return “true” in case of inconsistence mode between two common CAN channels.</p>	
Particularities and Limitations	
This API is only usable when switch “#define CAN_GET_STATUS STD_ON” is set by Generation tool.	

6.2.25 ApplCanInterruptDisable/Restore (None AUTOSAR API)

Prototype	
<pre>void ApplCanInterruptDisable(uint8 controller); void ApplCanInterruptRestore(uint8 controller);</pre>	
Parameter	
Controller	Controller for Interrupt lock
Return code	
-	
Functional Description	
Disable and Restore Can Interrupt by Application. Reason may be, that the CAN driver should not access the Interrupt controller for CAN interrupts or there are other restrictions (special security level)	
Particularities and Limitations	
<p>> This API is only usable when switch “#define CAN_INTLOCK” is set to “CAN_APPL” or “CAN_BOTH” by Generation tool.</p>	

6.2.26 Appl_CanOverrun / ApplCanFullCanOverrun (None AUTOSAR API)

Prototype	
<pre>void Appl_CanOverrun(uint8 controller); void Appl_CanFullCanOverrun(uint8 controller);</pre>	
Parameter	
Controller	Controller on which the overrun is detected
Return code	
-	
Functional Description	
This functions will be called if an overrun is detected.	
Particularities and Limitations	
<p>> This API is only usable when switch “#define CAN_OVERRUN_NOTIFICATION CAN_APPL” is set by Generation tool.</p> <p>> (There is also the possibility to deactivate this feature or call a DET instead)</p>	

6.2.27 Appl_CanCorruptMailbox (None AUTOSAR API)

Prototype	
<pre>void Appl_CanCorruptMailbox(uint8 Controller, Can_HwHandleType hwObjHandle)</pre>	
Parameter	
Controller	Controller on which the check failed
hwObjHandle	Hardware handle of defect mailbox
Return code	

-	
Functional Description	
Notify Application about a defect mailbox in CAN cell (while call of Can_InitController).	
Particularities and Limitations	
> This API is only used when feature is activated in Generation tool. (switch CAN_RAM_CHECK)	

6.2.28 Appl_CanRamCheckFailed (None AUTOSAR API)

Prototype	
uint8 Appl_CanRamCheckFailed(uint8 Controller)	
Parameter	
Controller	Controller on which the check failed
Return code	
uint8 action	Application decide by return value to activate or deactivate controller. CAN_DEACTIVATE_CONTROLLER CAN_ACTIVATE_CONTROLLER
Functional Description	
Notify Application about a defect CAN controller because of mailbox check before failed (while call of Can_InitController).	
Particularities and Limitations	
> This API is only used when feature is activated in Generation tool. ("CanRamCheck" -> switch CAN_RAM_CHECK)	

6.3 Services used by CAN

In the following table services provided by other components, which are used by the CAN are listed. For details about prototype and functionality refer to the documentation of the providing component.

Component	API
DET	Det_ReportError (see "Development Error Reporting")
DEM	Dem_ReportErrorStatus (see "Production Code Error Reporting")
EcuM	EcuM_CheckWakeup This function is called when Wakeup over CAN bus occur. EcuM_GeneratorCompatibilityError This function is called during the initialization, of the CAN Driver if the Generator Version Check or the CRC Check fails. (see [5])

Application (optional non AUTOSAR)	Appl_GenericPreCopy Appl_GenericConfirmation Appl_GenericPreTransmit ApplCanTimerStart/Loop/End Appl_CanRamCheckFailed, Appl_CanCorruptMailbox ApplCanInterruptDisable/Restore Appl_CanOverrun, Appl_CanFullCanOverrun For detailed description see Chapter 6.2
CANIF	CanIf_CancelTxNotification (non AUTOSAR) A special Software cancellation callback only used within Vector CAN driver CAN Interface bundle. CanIf_TxConfirmation Notification for a successful transmission. (see [4]) CanIf_CancelTxConfirmation Notification for a successful Tx cancellation. (see [4]) CanIf_RxIndication Notification for a message reception. (see [4]) CanIf_ControllerBusOff Bus Off notification function. (see [4]) CanIf_ControllerModeIndication MICROSAR4x only: Notification for mode successfully changed.
Os (MICROSAR4x)	OS_TICKS2MS_<counterShortName>() Os macro to get timebased ticks from counter. GetElapsedValue Get elapsed tick count. GetCounterValue Get tick count start.

Table 6-1 Services used by the CAN

7. Configuration

For CAN driver the attributes can be configured with configuration Tool “GENy”

The CAN driver supports pre-compile, link-time and post-build configuration.

For post-build systems, re-flashing the generated data can change some configuration settings.

For post-build and link-time configurations pre-compile settings are configured at compile time and therefore unchangeable at link or post-build time.

The following parameters are set by GENy configuration (see Chapter “Configuration with GENy”).

7.1 Pre-Compile Parameters

Some settings have to be available before compilation:

- > Version API (Can_GetVersionInfo() activation)
`#define CAN_VERSION_INFO_API STD_ON/STD_OFF`
- > DET (development error detection)
`#define CAN_DEV_ERROR_DETECT STD_ON/STD_OFF`
- > Hardware Loop Check (timeout monitoring)
`#define CAN_HARDWARE_CANCELLATION STD_ON/STD_OFF`
- > Polling modes: Tx confirmation, Reception, Wakeup, BusOff
`#define CAN_TX_PROCESSING CAN_INTERRUPT/ CAN_POLLING`
`#define CAN_RX_PROCESSING CAN_INTERRUPT/ CAN_POLLING`
`#define CAN_BUSOFF_PROCESSING CAN_INTERRUPT/ CAN_POLLING`
`#define CAN_WAKEUP_PROCESSING CAN_INTERRUPT/ CAN_POLLING`
`#define CAN_INDIVIDUAL_PROCESSING STD_ON/STD_OFF`
- > Multiplexed Tx (external PIA – by use multiple Tx mailboxes)
`#define CAN_MULTIPLEXED_TRANSMISSION STD_ON/STD_OFF`
- > Configuration Variant (define the configuration type when using post build variant)
`#define CAN_ENABLE_SELECTABLE_PB`
- > Use Generic Precopy Function (None AUTOSAR feature)
`#define CAN_GENERIC_PRECOPY STD_ON/STD_OFF`
- > Use Generic Confirmation Function (None AUTOSAR feature)
`#define CAN_GENERIC_CONFIRMATION STD_ON/STD_OFF`
- > Use Rx Queue Function (None AUTOSAR feature)
`#define CAN_RX_QUEUE STD_ON/STD_OFF`
- > Used ID type (standard/extended or mixed ID format)
`#define CAN_EXTENDED_ID STD_ON/STD_OFF`
`#define CAN_MIXED_ID STD_ON/STD_OFF`

- > Usage of Rx and Tx Full and BasicCAN objects (deactivate only when not using and to save ROM and runtime consumption)
`#define CAN_RX_FULLCAN_OBJECTS STD_ON/STD_OFF`
`#define CAN_TX_FULLCAN_OBJECTS STD_ON/STD_OFF`
`#define CAN_RX_BASICCAN_OBJECTS STD_ON/STD_OFF`
- > Use Multiple BasicCAN objects
`#define CAN_MULTIPLE_BASICCAN STD_ON/STD_OFF`
- > Optimizations
`#define CAN_ONE_CONTROLLER_OPTIMIZATION STD_ON/STD_OFF`
`#define CAN_DYNAMIC_FULLCAN_ID STD_ON/STD_OFF`
- > Usage of nested CAN interrupts
`#define CAN_NESTED_INTERRUPTS STD_ON/STD_OFF`
- > Use Multiple ECU configurations
`#define CAN_MULTI_ECU_CONFIG STD_ON/STD_OFF`
- > Use RAM Check (verify mailbox buffers)
`#define CAN_RAM_CHECK x`
- > Use Overrun detection
`#define CAN_OVERRUN_NOTIFICATION x`
- > Select MicroSar version
`#define CAN_MICROSAR_VERSION CAN_MSR40/CAN_MSR30`
- > Tx Cancellation of Identical IDs
`#define CAN_IDENTICAL_ID_CANCELLATION STD_ON/STD_OFF`

7.2 Link-Time Parameters

The library version of the CAN driver use following generated settings:

- > Maximum amount of used controllers and Tx mailboxes (has to be set for post-build variants at linktime)

7.3 Post-Build Parameters

Following settings are post-build data that can be changed for re-flashing:

- > Amount and usage of FullCAN Rx and Tx mailboxes
- > Used database (message information like ID, DLC)
- > Filters for BasicCAN Rx mailbox
- > Baud-rate settings
- > Module Start Address (only for post-build systems: The memory location for re-flashed data has to be defined)
- > Configuration ID (only for post-build systems: This number is used to identify the post-build data)

7.4 Configuration with GENy

The CAN driver is configured with the help of the configuration tool GENy.

To generate communication specific settings, like used message objects e.g. FullCAN Rx mailboxes, baud rate settings and hardware specific settings for CAN communication, the GENy tool is used. This chapter explains the usage of this tool.

Please refer to the online help of the tool for detailed information about the general usage. And see below for special settings of the hardware specifics.

7.4.1 Platform Settings

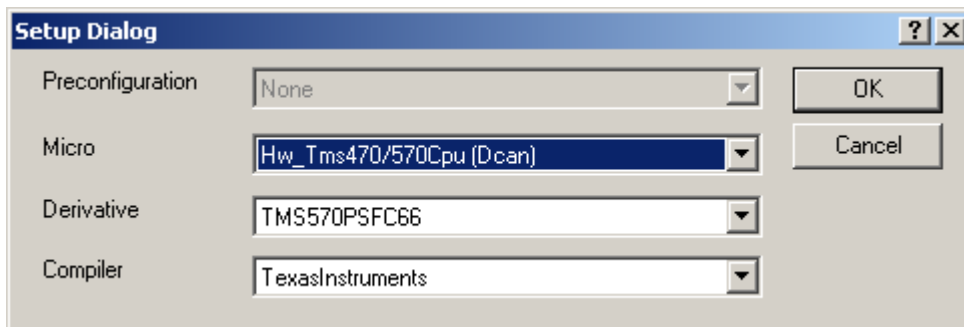


Figure 7-1 Platform settings

Attribute Name	Attribute is of type (Configuration Variant)	Value Type	Values	Description
Micro	pre-compile	Enum	Hw_Tms470/570Cpu(Dcan)	Select the hardware platform to use
Derivative	pre-compile	Enum	TMS470PSF761 TMS570PSF762 TMS570PSFC66 TMS570LS30316U	The specific derivative, based on the platform
Compiler	pre-compile	Enum	Texas Instruments	Compiler selection

Table 7-1 Platform Parameter description

7.4.2 Component Settings

ECU: CH0_Node0

- Components
 - Can_Tms470Dcan
 - CanIf
 - GenTool_GenyAsrBase
 - GenTool_GenyPluginAsrIfEcu
 - GenTool_GenyPluginConfigD
 - Hw_Tms470/570Cpu (Dcan)
 - NameDecorator
 - zBrs_EmbeddedRunTimeSys
- Tx Messages
- Rx Messages
- Tx Signals
- Rx Signals

Configurable Options	Can_Tms470Dcan
Miscellaneous	
Power Down Mode Select	Local
Hardware Loop Check	<input checked="" type="checkbox"/>
Enable Wakeup Support	<input checked="" type="checkbox"/> *
Identical ID cancellation	<input type="checkbox"/> *
Timeout	
Timeout Duration Factor	0x1388
Polling	
Tx Processing	Interrupt
Rx Processing	Interrupt
Busoff Processing	Interrupt
Wakeup Processing	Interrupt
AUTOSAR extension	
Generic Precopy	<input checked="" type="checkbox"/>
Generic Confirmation	<input checked="" type="checkbox"/>
Rx FullCAN Support	<input checked="" type="checkbox"/> *
Tx FullCAN Support	<input checked="" type="checkbox"/> *
Rx BasicCAN Support	<input checked="" type="checkbox"/> *
Hardware Loop Check by Application	<input type="checkbox"/> *
Report CAN_E_TIMEOUT DEM as DET 0x07 notification	<input type="checkbox"/> *
Support Mixed ID	<input type="checkbox"/> *
Type of Interrupt Function	Category 1
Get Status API	<input checked="" type="checkbox"/>
CAN Interrupt lock	Both
Optimize for one controller	<input type="checkbox"/> *
Dynamic FullCAN Tx ID	<input checked="" type="checkbox"/> *
Size of Hw HandleType	8bit
Partial Networking	<input type="checkbox"/> *
RAM check	Active
Overrun Notification	DET
Generic PreTransmit	<input checked="" type="checkbox"/>
Hw_Tms470DcanCpuCan	
Errata Dcan22 iterations	255*
Common	
Configuration Variant	Variant 3 (Post-build Configuration)
Version Info Api	<input checked="" type="checkbox"/>
Dev Error Detect	<input checked="" type="checkbox"/> *
Prod Error Detect	<input checked="" type="checkbox"/> *
User Config File	...
Postbuild Configuration	

Figure 7-2 Component Settings

Attribute Name	Configuration Variant	Value Type	Values	Description
Miscellaneous				
Hardware Loop Check	pre-compile	Bool	On / Off	Timeout monitoring to check for hardware problems inside a loop (possible endless loops because of hardware issues will be notified over DEM)
Timeout Duration Factor	pre-compile	Integer	Loop count	Maximum loop count for "Hardware Loop Check". Refer to loop descriptions set this value for your needs.
Timeout Counter ID (MICROSAR401 only)	pre-compile	Float	Time	Specifies the counter ID used in OS for 'Timeout Duration'
Multiplexed Transmission	pre-compile	Bool	On / Off	Activate to get multiplexed transmit objects for Tx BasicCAN object (3 send objects instead of 1) avoids external Id priority inversion
Wakeup Support	pre-compile	Bool	On / Off	Activate to Wake up over CAN bus by CAN driver
Polling				
Tx Processing	pre-compile	Enum	Interrupt Polling /	Activate to handle transmit messages over polling
Rx Processing	pre-compile	Enum	Interrupt Polling /	Activate to handle receive messages over polling
Busoff Processing	pre-compile	Enum	Interrupt Polling /	Activate to handle Bus Off event over polling
Wakeup Processing	pre-compile	Enum	Interrupt Polling /	Activate to handle Wakeup event over polling
Mainfunction bus off period	pre-compile	Float	Time	Period to call Can_MainFunction_BusOff() in seconds
Mainfunction wakeup period	pre-compile	Float	Time	Period to call Can_MainFunction_Wakeup() in seconds
Mainfunction read period	pre-compile	Float	Time	Period to call Can_MainFunction_Read() in seconds
Mainfunction write period	pre-compile	Float	Time	Period to call Can_MainFunction_Write() in seconds
AUTOSAR extension				
Generic PreCopy	pre-compile	Bool	On / Off	Generic Rx notification for all messages
Generic Confirmation	pre-compile	Bool	On / Off	Generic Tx confirmation for all messages

Attribute Name	Configuration Variant	Value Type	Values	Description
Generic PreTransmit	pre-compile	Bool	On / Off	A generic PreTransmit, common for all Tx messages will be called when this checkbox is enabled. Use this to change data or abort transmission right before the message will be send.
Rx FullCAN Support	pre-compile	Bool	On / Off	Use Rx FullCAN message objects (deactivate to reduce ROM and runtime consumption)
Tx FullCAN Support	pre-compile	Bool	On / Off	Use Tx FullCAN message objects (deactivate to reduce ROM and runtime consumption)
Rx BasicCAN Support	pre-compile	Bool	On / Off	Use Rx BasicCAN message objects (deactivate to reduce ROM and runtime consumption)
Use Nested CAN Interrupts	pre-compile	Bool	On / Off	CAN ISRs from different controllers can interrupt each other (only with higher priority). deactivate to reduce runtime consumption.
Secure Rx Buffer used	pre-compile	Bool	On / Off	Temporary buffer for received data will be located in a unused mailbox object. This is to secure receive data by not locate it to common RAM.
Hardware Loop Check by Application	pre-compile	Bool	On / Off	Enable this when you like to handle "Hardware Loop Check" by your own. Using API functions "ApplCanTimerStart()", „ApplCanTimerEnd()" and „ApplCanTimerLoop()"
Report CAN_E_TIMEOUT DEM as DET 0x07 notification	pre-compile	Bool	On / Off	Enable this when you like to report CAN_E_TIMEOUT ("Hardware Loop Check") to DET instead of DEM (does only work when DET is activated)
Individual Processing	pre-compile	Bool	On / Off	Enable this when you like to configure mailbox specific polling/interrupt processing. This feature is only available with "High End" license.
Support Mixed ID	pre-compile	Bool	On / Off	Force CAN driver to handle Mixed ID (standard and extended ID) at pre-compile-time to expand the ID type later on.
Multiple BasicCAN Objects	pre-compile	Bool	On / Off	Support Multiple BasicCAN objects. This gives the possibility to use multiple Filters and optimize acceptance Filtering as well as avoid overrun. This feature is only available with "High End" license.

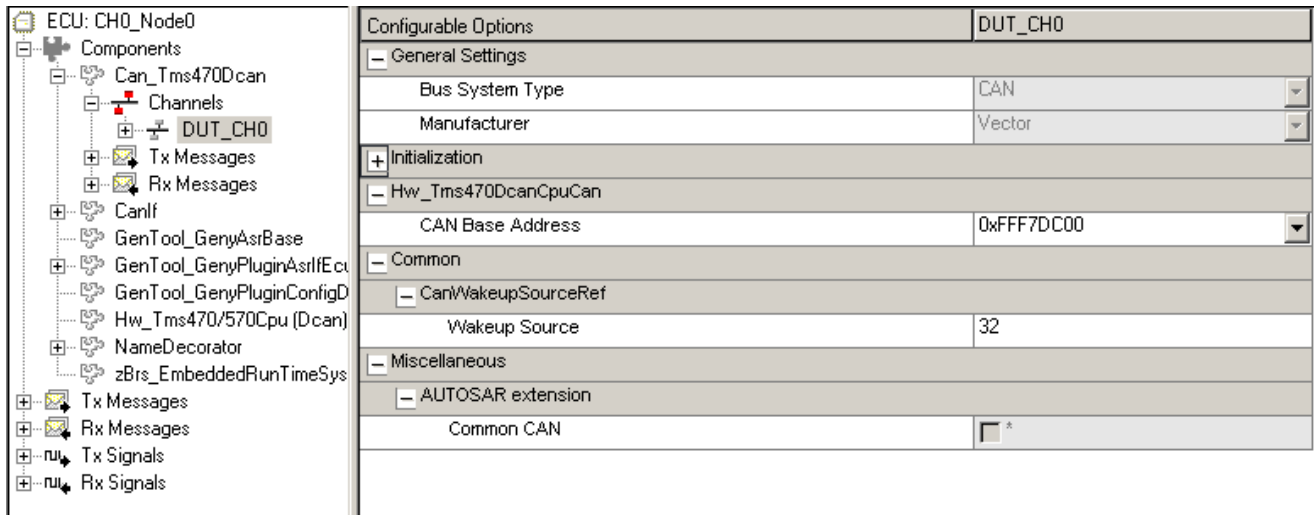
Attribute Name	Configuration Variant	Value Type	Values	Description
Optimize for one controller	pre-compile	Bool	On / Off	Activate this for 1 controller systems when you never will expand to multi-controller. So that the CAN driver works more efficient
Dynamic FullCAN Tx ID	pre-compile	Bool	On / Off	Always write FullCAN Tx ID within Can_Write() API function. Deactivate this to optimize code when you do not use FullCAN Tx objects dynamically.
Size of Hw HandleType	pre-compile	Enum	8bit 16bit	Support 8bit Hardware Handles as define in AutoSar 3.01 specified. Or change to 16bit like specified in AutoSar 4.0.
Partial Networking	pre-compile	Bool	On / Off	Activate this when you like to support a partial network that will be woken up by special NM message. This feature have to be enabled if one of the controllers will wake up by partial network transceiver.
Get Status API	pre-compile	Bool	On / Off	Support Can_GetStatus() API that return the hardware states: CAN_STATUS_STOP, CAN_STATUS_INIT, CAN_STATUS_INCONSISTENT, CAN_STATUS_WARNING, CAN_STATUS_PASSIVE, CAN_STATUS_BUSOFF, CAN_STATUS_SLEEP
Type of Interrupt Function	Link-time	Enum	Category 1 Category 2 VoidFuncVoid	Category 1: Interrupt Function has to be added to the interrupt vector table unless this is not already done automatically by a compiler pragma. Category 2: Interrupt Function is defined with ISR() define. VoidFuncVoid: Interrupt Function is declared as void ISR(void) and has to be called in case of a CAN Interrupt.
CAN Interrupt lock	pre-compile	Enum	CANDriver Application Both	Disable and Restore Can Interrupt by CAN Driver or Application or CAN Driver and Application. Reason maybe, that the CAN driver should not access the Interrupt controller for CAN interrupts or there are other restrictions (special security level) or Application has to care about additional Interrupts like Wakeup.

Attribute Name	Configuration Variant	Value Type	Values	Description
Overrun Notification	pre-compile	Enum	None DET Application	Overrun detection activation and selection of the Notification function. None: No Overrun detection. DET: will be called for Overrun. Application: will be called.
RAM check	pre-compile	Enum	None Active Mailbox Notification	RAM check of mailbox buffers None: No RAM check Active: Active RAM check (global notification) Mailbox Notification: Active RAM check (global + mailbox notification)
Hardware (Transmit) Cancellation	pre-compile	Bool	On / Off	Enable/disable the usage of 'transmit cancellation' out of hardware mailboxes, so that the canceled message won't be send. (avoid internal priority inversion)
Identical ID cancellation	pre-compile	Bool	On / Off	'Hardware Transmit Cancellation' also cancel messages with same identifier (newer data will be send) out of hardware mailboxes. otherwise only lower prior identifiers will be cancelled. (older data will be send)
Rx Queue				
Rx Queue	pre-compile	Bool	On / Off	Enable this when you like to handle Rx messages in polling context. (Shorten interrupt latency time). This feature is only available with "High End" license.
Size	pre-compile	Bool	On / Off	Size of FIFO in which the Message information is buffered.
Common				
Configuration Variant	pre-compile	Enum	Pre-Compile, Link-Time, Post build	Select your project type
Version Info API	pre-compile	Bool	On / Off	Activate function to get version information
Dev Error Detection	pre-compile	Bool	On / Off	Activate to get information about possible conflicts (refer to DET module description)
Prod Error Detection	pre-compile	Bool	On / Off	Activate to get information about possible conflicts in a running system (refer to DEM module description)

Attribute Name	Configuration Variant	Value Type	Values	Description
User Config File	pre-compile	String	File name and path	This file will be included in the Can_cfg.h for special settings
Post build configuration				
Module Start Address (Post-build only)	post-build	Integer	Address	Address to Flash the data
Max Nr. Of Tx Objects (Post-build only)	Link-time	Integer	Amount	Amount of maximum used transmit objects over all controllers (used for RAM variables). One object for BasicCAN Tx and one for each FullCAN Tx. Add all objects over all used Channels for this value.
Max CAN Controller (Post-build only)	Link-time	Integer	Amount	Amount of maximum used controllers (used for RAM variables)
Configuration ID (Post-build only)	post-build	Integer	Identity	A special value to identify this configuration in the Flash.

Table 7-2 Component Parameter description

7.4.3 Controller (Channel) Settings



The screenshot displays the configuration interface for the 'DUT_CH0' component. On the left, a tree view shows the component hierarchy under 'ECU: CH0_Node0', including 'Components', 'Can_Tms470Dcan', 'Channels', 'Tx Messages', 'Rx Messages', 'CanIf', and various GenTool and Hw_Tms470/570Cpu (Dcan) components. The right pane shows the 'Configurable Options' for 'DUT_CH0', organized into several sections:

- General Settings:**
 - Bus System Type: CAN
 - Manufacturer: Vector
- Initialization:**
 - Hw_Tms470DcanCpuCan:
 - CAN Base Address: 0xFFFF7DC00
- Common:**
 - CanWakeupSourceRef:
 - Wakeup Source: 32
- Miscellaneous:**
 - AUTOSAR extension:
 - Common CAN: ☐ *

Attribute Name	Attribute is of type (Configuration Variant)	Value Type	Values	Description
Miscellaneous				
Wakeup Source ID / Wakeup Source Ref <small>(Reference is only available if EcuM is available in GENy)</small>	Link-time	Integer	Source	Insert the Wakeup Source value compatible to ECU State Manager.
Enable Wakeup Support	Link-time	Bool	On / Off	MICROSAR4x only: Activation of Wakeup over CAN bus.
AUTOSAR extension				
Partial Networking	Post-build	Bool	On / Off	Activate this when you there is a partial network transceiver on this controller active and used.

Table 7-2 Controller Parameter description

7.4.4 Init Structure Settings

Configurable Options		DUT_CH0
[-] General Settings		
Bus System Type	CAN	
Manufacturer	Vector	
[-] Initialization		
[-] Init Structures		Add
[-] Init Structure		Delete
Bit Timing Reg. 0	0x1f	
Bit Timing Reg. 1	0x4d	
Bit Timing Reg. 2	0x0*	
Acceptance Filter Configuration	...	
Bustiming Configuration	...	
[-] Init Structure		Delete
Bit Timing Reg. 0	0x6f	
Bit Timing Reg. 1	0x3a*	
Bit Timing Reg. 2	0x0*	
Acceptance Filter Configuration	...	
Bustiming Configuration	...	
[-] Init Structure		Delete
Bit Timing Reg. 0	0x7	
Bit Timing Reg. 1	0x3a*	
Bit Timing Reg. 2	0x0*	
Acceptance Filter Configuration	...	
Bustiming Configuration	...	

Figure 7-2 Init Structure Dialog

Add Init Structures to setup multiple baud rate and filter settings that can be selected at Initialization phase of the ECU.

MicroSar3 only: This structure can be selected with `Can_InitStruct()` before call `Can_InitController()`.

7.4.4.1 Setup the Filter by pressing the “Acceptance Filter Configuration” Button.

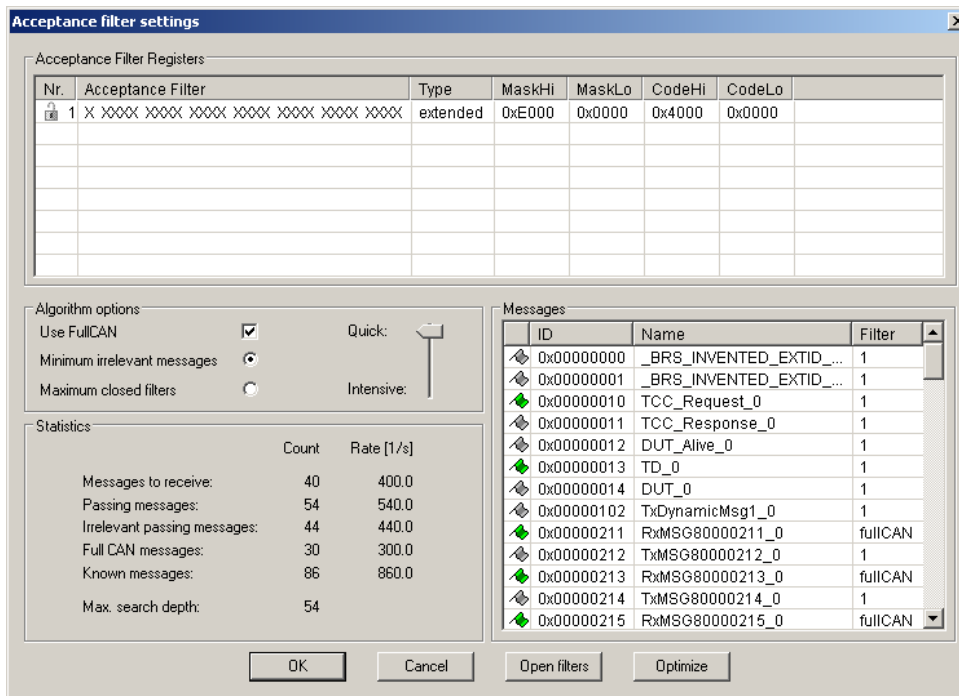
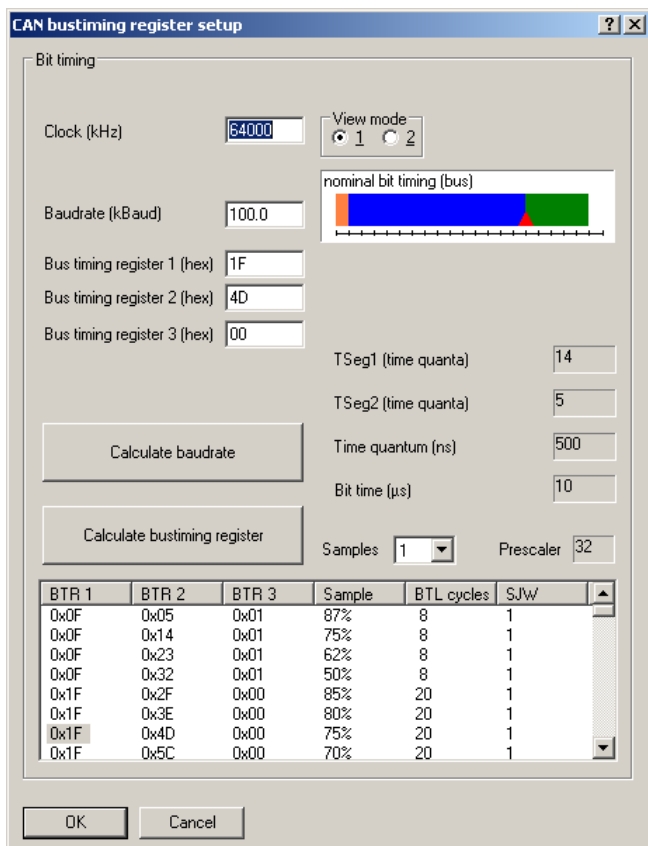


Figure 7-3 Setup Filter Dialog

Attribute Name	Attribute is of type (Configuration Variant)	Value Type	Values The default value is written in bold	Description
Setup				
Acceptance Filter	Post-build	Integer	Mask, code	Each ID –bit is represented by a “0/1/X”, means must match a “0” or “1” or must not match “X”
Open filters		Action	-	Open the filter complete to receive all IDs
Optimize		Action	-	Open the filter automatic to just receive the IDs in database. “Use FullCAN” try to put as much messages in FullCAN objects to better optimize filters
Messages-Window			Messages that will be received via ranges (like NM messages) will have always a grey flag in view. They will be included in “Irrelevant passing messages” amount.	Open the filter automatic to just receive the IDs in database. “Use FullCAN” try to put as much messages in FullCAN objects to better optimize filters

Table 7-3 Filter Parameter description

7.4.4.2 Setup the Baud rate by pressing the “Bus timing Configuration” Button



The dialog box titled "CAN bustiming register setup" contains the following fields and controls:

- Bit timing** section:
 - Clock (kHz): 64000
 - Baudrate (kBaud): 100.0
 - Bus timing register 1 (hex): 1F
 - Bus timing register 2 (hex): 4D
 - Bus timing register 3 (hex): 00
- View mode**: Radio buttons for 1 (selected) and 2.
- nominal bit timing (bus)**: A graphical representation of a bit period with segments in orange, blue, and green.
- Calculate baudrate** button.
- Calculate bustiming register** button.
- TSeg1 (time quanta)**: 14
- TSeg2 (time quanta)**: 5
- Time quantum (ns)**: 500
- Bit time (μs)**: 10
- Samples**: 1 (dropdown)
- Prescaler**: 32
- Table**:

BTR 1	BTR 2	BTR 3	Sample	BTL cycles	SJW
0x0F	0x05	0x01	87%	8	1
0x0F	0x14	0x01	75%	8	1
0x0F	0x23	0x01	62%	8	1
0x0F	0x32	0x01	50%	8	1
0x1F	0x2F	0x00	85%	20	1
0x1F	0x3E	0x00	80%	20	1
0x1F	0x4D	0x00	75%	20	1
0x1F	0x5C	0x00	70%	20	1
- OK** and **Cancel** buttons at the bottom.

Figure 7-4 Baud Rate Dialog

Attribute Name	Attribute is of type (Configuration Variant)	Value Type	Values	Description
Clock	Post-build	Integer	CAN clock	Set the frequent of the CAN cell clock.
Baudrate	Post-build	Integer	Baud rate	Set baud rate to be used for this channel
Calculate		Action	-	It is possible to calculate possible hardware register settings out of baud rate or vice versa.
Sample, BTL cycles, SJW	Post-build	Select	Sample Point	Select the sample point and sync phase related to your bus physics

Table 7-4 Baud rate Parameter description

7.5 Configuration with da DaVinci Configurator

See Online help within DaVinci Configurator and BSWMD file for parameter settings.

8. AUTOSAR Standard Compliance

8.1 Limitations / Restrictions

- > No multiple AUTOSAR CAN driver allowed in the system.
- > No support for L-PDU callout (AUTOSAR 3.2.1), but support 'Generic Precopy' instead
- > No support for multiple read and write period configuration (AUTOSAR 3.2.1)
- > Configurations with no Tx Basic CAN are not supported

8.2 Vector Extensions

Refer to Chapter "[Features](#)" listed under "**AUTOSAR extensions**"

9. Glossary and Abbreviations

9.1 Glossary

Term	Description
GENy	Generation tool for CANbedded and MICROSAR components
High End (license)	Product license to support an extended feature set (see Feature table)

Table 9-1 Glossary

9.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
DEM	Diagnostic Event Manager
DET	Development Error Tracer
ECU	Electronic Control Unit
HIS	Hersteller Initiative Software
ISR	Interrupt Service Routine
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution) 3,3x = AUTOSAR version 3 401 = AUTOSAR version 4.0.1 403 = AUTOSAR version 4.0.3 4x = AUTOSAR version 4.x.x
SWS	Software Specification
Common CAN	Connect two physical peripheral channels to one CAN bus (to increase the amount of FullCAN)
Hardware Loop Check	Timeout monitoring for possible endless loops.

Table 9-2 Abbreviations

10. Contact

Visit our website for more information on

- > News
- > Products
- > Demo software
- > Support
- > Training data
- > Addresses

www.vector.com