

MICROSAR CAN Driver

Technical Reference

CANoe / VTTCan

Version 2.08.00

Authors	Holger Birke
Status	Released

1 Document Information

1.1 History

Platforms

Author	Date	Version	Remarks
Holger Birke	2006-06-29	1.1	First AUTOSAR release
Holger Birke	2007-02-15	1.2	Update to Post build
Holger Birke	2007-07-25	1.3	Update to AUTOSAR specification 2.1
Holger Birke	2007-12-14	1.4	Add extended/mixed id support
Holger Birke	2008-03-03	1.5	Update to ASR 3
Holger Birke	2008-05-14	1.6	Minor Review changes
Holger Birke	2008-08-13	1.7	Update Tool screenshots for new release
Holger Birke	2008-10-24	1.8	Update Tool screenshots (ASR naming)
Holger Birke	2009-06-08	1.9	Update Core (Compiler Abstraction) + Tool screenshots for new release
Holger Birke	2009-07-20	1.10	Update Core/template + Tool screenshots for new release
Holger Birke	2011-03-15	2.00	MICROSAR 4 support added Update Core/template + Tool screenshots for new release
Holger Birke	2011-12-16	2.01	Update Core/template + Tool screenshots for new release
Holger Birke	2012-02-08	2.01.01	Fix some minor findings
Holger Birke	2012-08-10	2.02.00	MICROSAR 4.0.3 support (Core 4.03.00 – R5)
Holger Birke	2012-08-10	2.03.00	Update Core/template (Core 4.05.00 – R7)
Holger Birke	2013-10-14	2.04.00	Update Core/template (Core 4.06.00 – R8)
Holger Birke	2014-10-24	2.05.00	Update Core/template (Core 5.00.00 – R11)
Holger Birke	2015-06-30	2.05.01	ESCAN00083619 RAM check description improvement
Holger Birke	2015-07-10	2.07.00	Update Core/template (Core 5.02.00 – R13)
Holger Birke	2015-11-27	2.07.01	Update AutoSar deviation table (R14)
Holger Birke	2017-06-07	2.08.00	Update SilentMode (R19)

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	3
1.3	Scope of the Document	3
2	Hardware Overview	7
3	Introduction	8
3.1	Architecture Overview	9
4	Functional Description	11
4.1	Features	11
4.2	Initialization	14
4.3	Communication	14
4.4	States / Modes	16
4.5	Re-Initialization	17
4.6	CAN Interrupt Locking	17
4.7	Main Functions	17
4.8	Error Handling	18
4.9	Common CAN	22
4.10	Hardware Specific	22
5	Integration	23
5.1	Scope of Delivery	23
5.2	Include Structure	24
5.3	Critical Sections	24
5.4	Compiler Abstraction and Memory Mapping	26
5.5	Hardware Specific Hints	27
6	API Description	28
6.1	Interrupt Service Routines provided by CAN	28
6.2	Services provided by CAN	30
6.3	Services used by CAN	53
7	Configuration	55
7.1	Pre-Compile Parameters	55
7.2	Link-Time Parameters	56
7.3	Post-Build Parameters	56
7.4	Configuration with da DaVinci Configurator	57

8	AUTOSAR Standard Compliance	58
8.1	Limitations / Restrictions.....	58
8.2	Vector Extensions	58
9	Glossary and Abbreviations	59
9.1	Glossary.....	59
9.2	Abbreviations	59
10	Contact.....	60

Tables

Table 1-1	History of the Document.....	2
Table 1-2	Reference Documents.....	3
Table 2-1	Supported Hardware Overview	7
Table 4-1	Supported features.....	14
Table 4-2	Hardware mailbox layout.....	15
Table 4-3	Errors reported to DET	18
Table 4-4	API from which the Errors are reported	19
Table 4-5	Errors reported to DEM	20
Table 4-6	Hardware Loop Check.....	21
Table 5-1	Static files.....	23
Table 5-2	Generated files.....	23
Table 5-3	Critical Section Codes.....	25
Table 5-4	Compiler abstraction and memory mapping	26
Table 6-1	Can_InitMemory	31
Table 6-2	Can_Init.....	31
Table 6-3	Can_InitController	32
Table 6-4	Can_InitController	33
Table 6-5	Can_ChangeBaudrate.....	33
Table 6-6	Can_CheckBaudrate.....	34
Table 6-7	Can_SetBaudrate.....	35
Table 6-8	Can_InitStruct	35
Table 6-9	Can_GetVersionInfo.....	36
Table 6-10	Can_GetStatus.....	37
Table 6-11	Can_SetControllerMode.....	38
Table 6-12	Can_ResetBusOffStart	38
Table 6-13	Can_ResetBusOffEnd	39
Table 6-14	Can_Write	40
Table 6-15	Can_CancelTx	40
Table 6-16	Can_SetMirrorMode	41
Table 6-17	Can_CheckWakeup	41
Table 6-18	Can_DisableControllerInterrupts	42
Table 6-19	Can_EnableControllerInterrupts	42
Table 6-20	Can_MainFunction_Write	43
Table 6-21	Can_MainFunction_Read.....	43
Table 6-22	Can_MainFunction_BusOff	44
Table 6-23	Can_MainFunction_Wakeup	44
Table 6-24	Can_MainFunction_Mode	45
Table 6-25	Appl_GenericConfirmation	46

Table 6-26	Appl_GenericConfirmation	47
Table 6-27	Appl_GenericPreTransmit	47
Table 6-28	ApplCanTimerStart	48
Table 6-29	ApplCanTimerLoop	48
Table 6-30	ApplCanTimerEnd	49
Table 6-31	ApplCanInterruptDisable	50
Table 6-32	ApplCanInterruptRestore	50
Table 6-33	Appl_CanOverrun	51
Table 6-34	Appl_CanFullCanOverrun	51
Table 6-35	Appl_CanCorruptMailbox	52
Table 6-36	Appl_CanRamCheckFailed	53
Table 6-37	Services used by the CAN	54
Table 9-1	Glossary	59
Table 9-2	Abbreviations	59

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
CANoeEmu	Visual-C	Vector	-

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 Release:	AUTOSAR	3 and 4
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])
AR Version:	CAN_AR_RELEASE_MAJOR_VERSION CAN_AR_RELEASE_MINOR_VERSION CAN_AR_RELEASE_REVISION_VERSION	AUTOSAR Release version BCD coded
SW Version:	CAN_SW_MAJOR_VERSION CAN_SW_MINOR_VERSION CAN_SW_PATCH_VERSION	MICROSAR RTE version BCD coded

* 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.

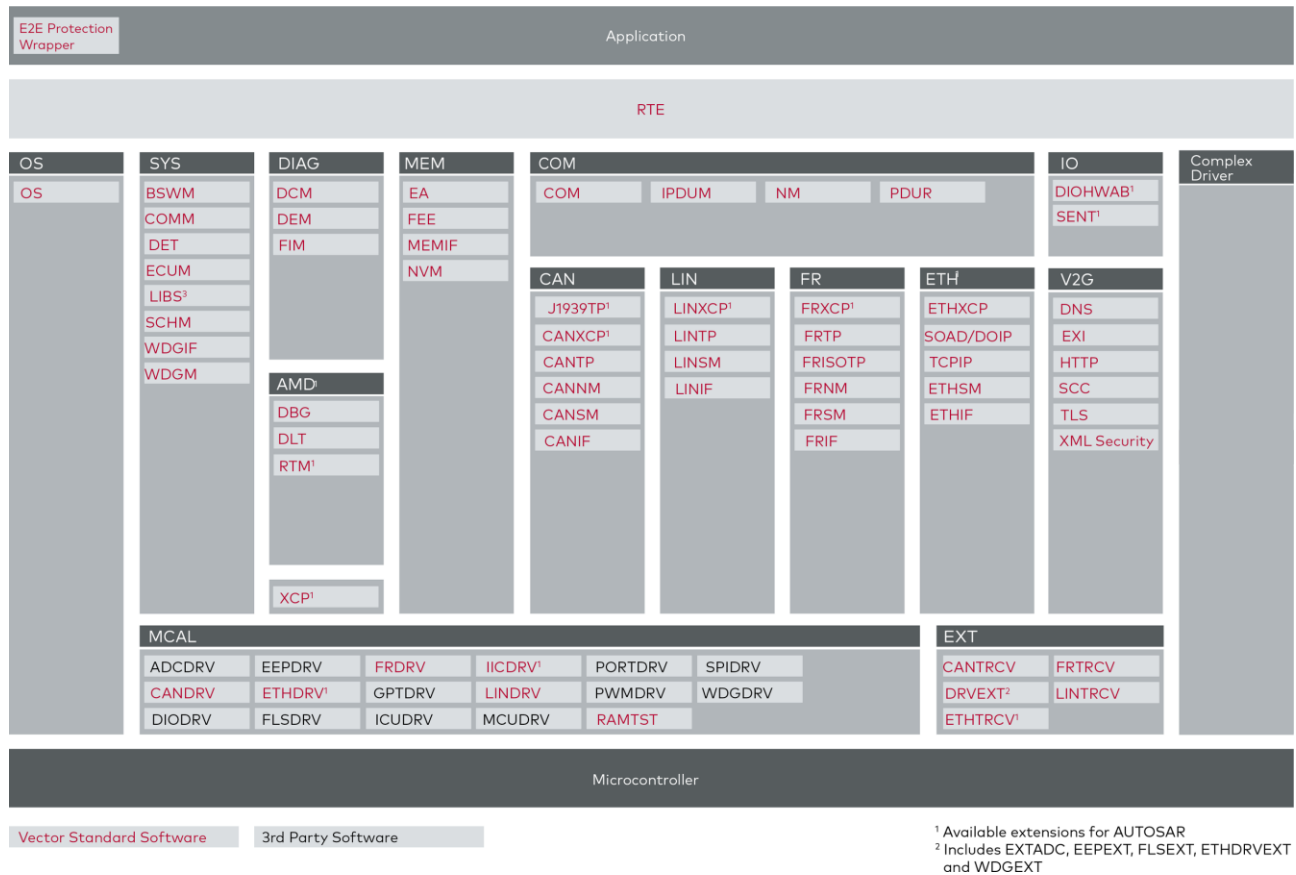


Figure 3-1 AUTOSAR 3.x Architecture Overview

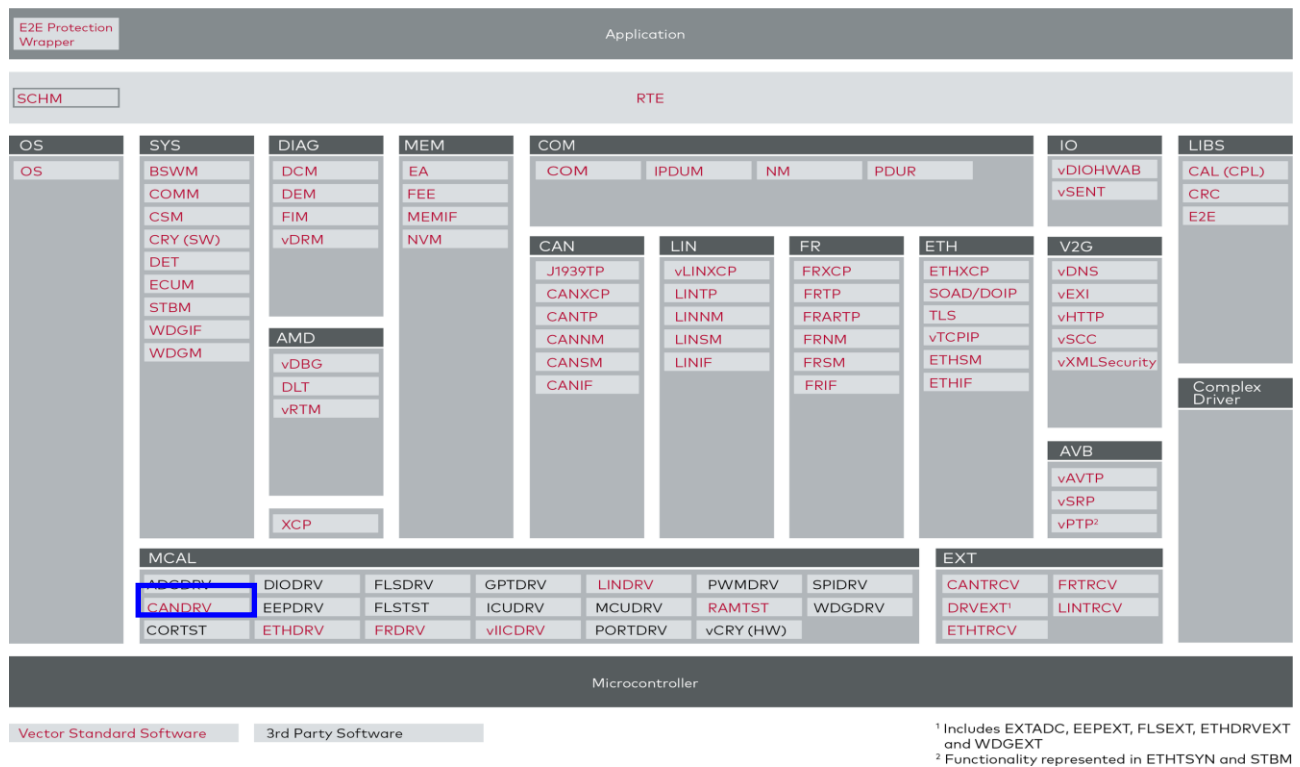


Figure 3-2 AUTOSAR architecture

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

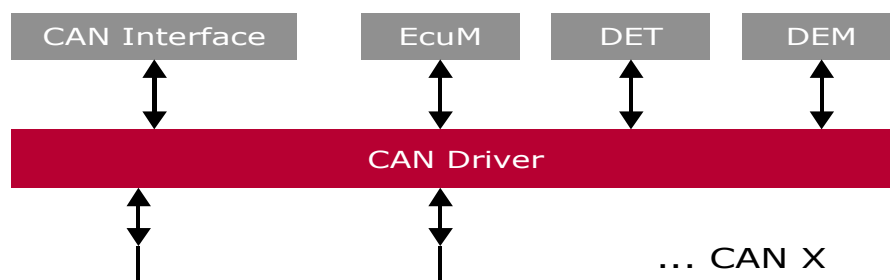


Figure 3-3 Interfaces to adjacent modules of the CAN

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	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.	■
Silent mode	Support SilentMode where the controller only listen passive.	■
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.	250/500
Rx FullCAN	Separate mailbox for special Rx message used.	■
Maximum amount	Available amount of mailboxes.	250/500
Rx BasicCAN	Standard mailbox to receive CAN frames (depending on hardware, FIFO or shadow buffer supported).	■
Maximum amount	Available amount of BasicCAN objects.	250/500
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).	16
Cancellation of TX objects	Support of TX Cancellation (out of hardware). Avoid internal priority inversion.	■
Identical ID cancellation	TX Cancellation also for identical IDs.	■ **
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).	■
CAN FD Mode1	FD frames with baudrate switch supported (TX and RX).	■
CAN FD Mode2	FD frames up to 64 data bytes 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).	■
Multiple Basic CAN	Support Multiple BasicCAN objects. This give the possibility to use multiple Filters and optimize acceptance Filtering as well as avoid overrun.	■
Rx Queue	Support Rx Queue. This give the possibility to buffer received data in interrupt context but handle it asynchronous in polling task.	■

Secure Rx Buffer used	Special hardware buffer used to temporary save received data.	<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"/>
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	■
Size of HW Handle Type	Support 8bit or 16bit Hardware Handles	■
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	<input type="checkbox"/>
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 / Multiple Identity	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 License only ** Only Emulated
 + CanoeEmu only ++ VTT only

4.2 Initialization

`Can_Init()` has to be called to initialize the CAN driver at power on and sets controller independent initialization 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
0 – n	TX FullCAN	0-500	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 BasicCAN	1 or 3 (depend on Multiplexed TX feature)	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(5) - m	Unused	0-500	These objects are not used. It depends on the configuration of receive and transmit objects if unused objects are available.
m+1 – o	Rx FullCAN	0-500	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 – p	Rx BasicCAN	1-500	All other CAN messages are received via the BasicCAN message object. Each BasicCAN message object consists of 1 message buffer and 1 Filter. If using "Multiple BasicCAN" feature the amount of Used BasicCAN can be configured (use also this feature to select no BasicCAN)

Table 4-2 Hardware mailbox layout

The "CanObjectId" (ECUc parameter) numbering is done in following order: FullCAN TX, BasicCAN TX, Unused, FullCAN Rx, BasicCAN Rx (like shown above). "CanObjectId's" for next controller begin at end of last controller. Gaps in "CanObjectId" for unused mailboxes may occur.

4.3.2 Mailbox Processing Order

The hardware mailbox will be processed in following order:

Object Type	Order / priority to send or receive
All Mailboxes (FullCAN and BasicCAN Rx and Tx)	FIFO (only emulated FullCANs)

In Case of Polling and Interrupt, FullCANs will be processed before BasicCANs.

The order between Rx and Tx mailboxes depends on the call order of the polling tasks or the interrupt context and cannot be guaranteed.

The Rx Queue will work like a FIFO filled with the above mentioned method.

4.3.3 Acceptance Filter for BasicCAN

The hardware acceptance filter of the BasicCAN object will be emulated in software. The filter in CANoe may remain full open and the CAN driver reject the unused messages according to the filter settings in the CAN driver. One receive mailbox will be used for each filter (standard or extended ID). For mixed ID used case please configure 2 Filters, one as standard and one as extended.

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).

4.3.4 Remote Frames

Remote Frames rejected in Hardware	Remote Frames rejected in Software
no	no

Remote Frames cannot be filtered for this platform they will be received like normal message.

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` (only emulated)

`CAN_T_WAKEUP`

MICROSAR4 only: Notification of mode change may occur asynchronous by notification `CanIf_ControllerModeIndication()`

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

The stop mode is an emulated mode for this platform and do not really disconnect from CAN bus. Due to that Acknowledge will be send for incoming messages.

4.4.3 Sleep Mode

The Sleep mode and the Wakeup event are only emulated. This means that a Acknowledge will be send, the controller just remember his state. A received message will cause a wakeup-event and is also received (notification).

`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 this mode.

If Wakeup support is deactivated by configuration, the controller went into stop mode instead of sleep.

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

4.4.5 Silent Mode

Support API (`Can_SetSilentMode()`) to switch into 'SilentMode' where the controller does not take part on BUS communication (no ACK) but can listen for messages.

Refer to ISO 11898 bus monitoring.

Canoe as an emulation does not support a internal rerouting so if there is no other node that acknowledge the bus no message will be received.

Rx error counter will be increased until passive for each not acknowledged message.

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.

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.

MICROSAR4 only: Can_MainFunction_Mode() called by upper layer to poll asynchronous mode transition notifications.

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
CAN_E_PARAM_POINTER	API gets an illegal pointer as parameter.
CAN_E_PARAM_HANDLE	API get an illegal handle as parameter
CAN_E_PARAM_DLC	API get an illegal DLC as parameter
CAN_E_PARAM_CONTROLLER	API get an illegal controller as parameter
CAN_E_UNINIT	Driver API is used but not initialized
CAN_E_TRANSITION	Transition for mode change is illegal
CAN_E_DATA_LOST (value: 0x07, AutoSar extension)	Rx overrun (overwrite) detected
CAN_E_PARAM_BAUDRATE (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_SET_BR_ID	Can_SetBaudrate()
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

This platform perform overwrite detection for FullCAN and BasicCAN receive mailbox.

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

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 Event ID and Event Status given as parameter for calling the DEM. This callout may occur in the context of different API calls (see Chapter “Hardware Loop Check”).

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 source	Name /	Short Description
kCanLoopInit		Only emulated loop (no real issue), will run in context of Can_InitController().
kCanLoopRx		<ul style="list-style-type: none"> - Loop to detect inconsistent data while message reception. - Call context: Can_MainfunctionRead() or Rx Interrupt. - Expected duration: 1 - Action after timeout: None. Received message will be ignored and the driver is ready for new reception.
kCanLoopStart		<p>MICROSAR3:</p> <ul style="list-style-type: none"> - Used while transition in mode 'START'. - Call context: Can_SetControllerMode() - Expected duration: 1 - Action after timeout: None (only emulation) <p>MICROSAR4:</p> <p>Used for short time mode transition blocking (short synchronous timeout). Same value for kCanLoopStart, kCanLoopStop, kCanLoopSleep and kCanLoopWakeup. No Issue when timeout occur.</p>

kCanLoopStop	<p>MICROSAR3:</p> <ul style="list-style-type: none"> - Used while transition in mode 'STOP'. - Call context: Can_SetControllerMode() - Expected duration: 1 - Action after timeout: None (only emulation) <p>MICROSAR4:</p> <p>Used for short time mode transition blocking (short synchronous timeout). Same value for kCanLoopStart, kCanLoopStop, kCanLoopSleep and kCanLoopWakeup. No Issue when timeout occur.</p>
kCanLoopSleep	<p>MICROSAR3:</p> <ul style="list-style-type: none"> - Used while transition in mode 'SLEEP'. - Call context: Can_SetControllerMode() - Expected duration: 1 - Action after timeout: None (only emulation) <p>MICROSAR4:</p> <p>Used for short time mode transition blocking (short synchronous timeout). Same value for kCanLoopStart, kCanLoopStop, 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: 1 - Action after timeout: None (only emulation) <p>MICROSAR4:</p> <p>Used for short time mode transition blocking (short synchronous timeout). Same value for kCanLoopStart, kCanLoopStop, kCanLoopSleep and kCanLoopWakeup. No Issue when timeout occur.</p>

Table 4-6 Hardware Loop Check

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() or a wakeup event occurs.

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

4.9 Common CAN

Common CAN connect 2 hardware CAN channels to one logical controller. This allows configuring more FullCAN mailboxes. The second hardware channel is used for Rx FullCAN mailboxes.

The filter mask of the BasicCAN should exclude the message received by the FullCAN messages of the second CAN Controller. This means each message ID must be received on one CAN hardware channel only. The filter optimization takes care about this when common CAN is activated.

For configuration of Common CAN specific settings in generation tool see chapter 7.6.2.



Please note

Only one Transceiver (Driver) has to be used for this two Common CAN hardware channels (connect TX and RX lines).

Reason: Upper layers only know one Controller for this 2 hardware channel Common CAN and therefore only one Transceiver can be handled.

4.10 Hardware Specific

CANoeEmu CAN driver is an emulation driver so it may react in some cases different from a real hardware. In special see STOP mode.

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
(+) Can_Local.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.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. (this is a project specific file and may not exist)
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

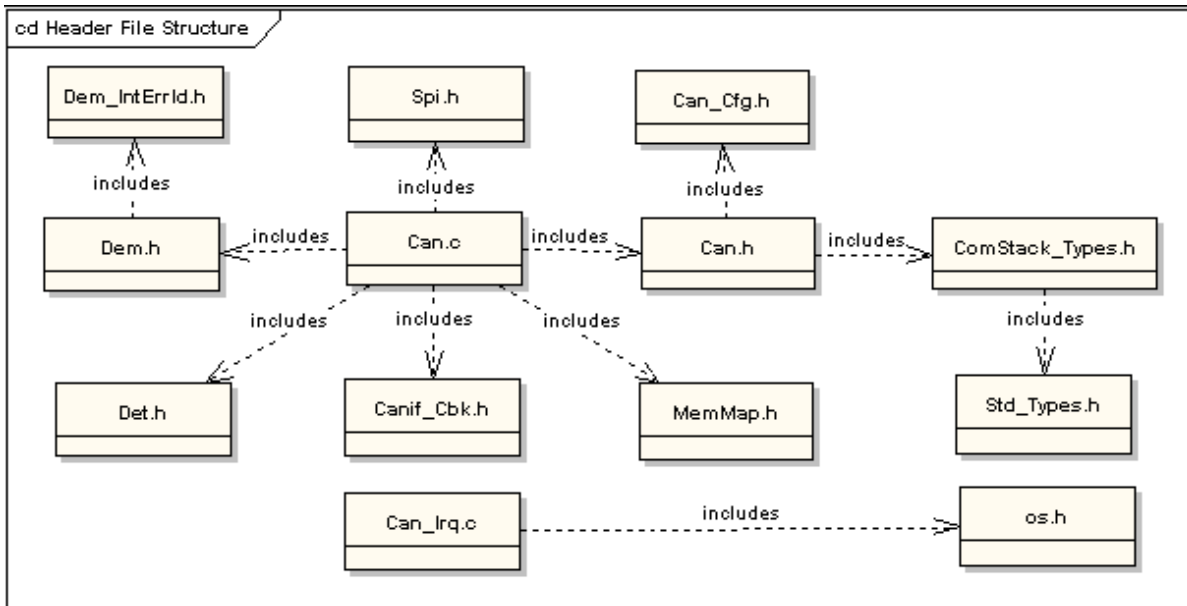


Figure 5-1 Include Structure (AUTOSAR)

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 may be included by Can.c.
- 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_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 ■ 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 ■ 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 medium ■ 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 ■ No API call of other BSW inside. ■ Disable Global Interrupts - or - Disable all CAN interrupts.
CAN_EXCLUSIVE_AREA_5	<p>Using inside wakeup handling to secure state transition. (Only in wakeup polling mode)</p> <ul style="list-style-type: none"> ■ Duration is short ■ Call to DET inside. ■ Disable global interrupts (do not use CAN interrupt locks here)
CAN_EXCLUSIVE_AREA_6	<p>Using inside Can_SetControllerMode() and BusOff to secure state transition.</p> <ul style="list-style-type: none"> ■ Duration is medium ■ 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_CANCELL	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.

CAN_ APPL_*: access to higher layers.

5.5 Hardware Specific Hints

BusName given by configuration have to be the same name as inside CANoe configuration for BusName to map the logical controller to a specified CAN cap

6 API Description

6.1 Interrupt Service Routines provided by CAN

Depend on the settings in Tools component Hw_CanoeemuCpu, the interrupt routine is given by the driver or by Operating System. (Selection below, not MICROSAR403)

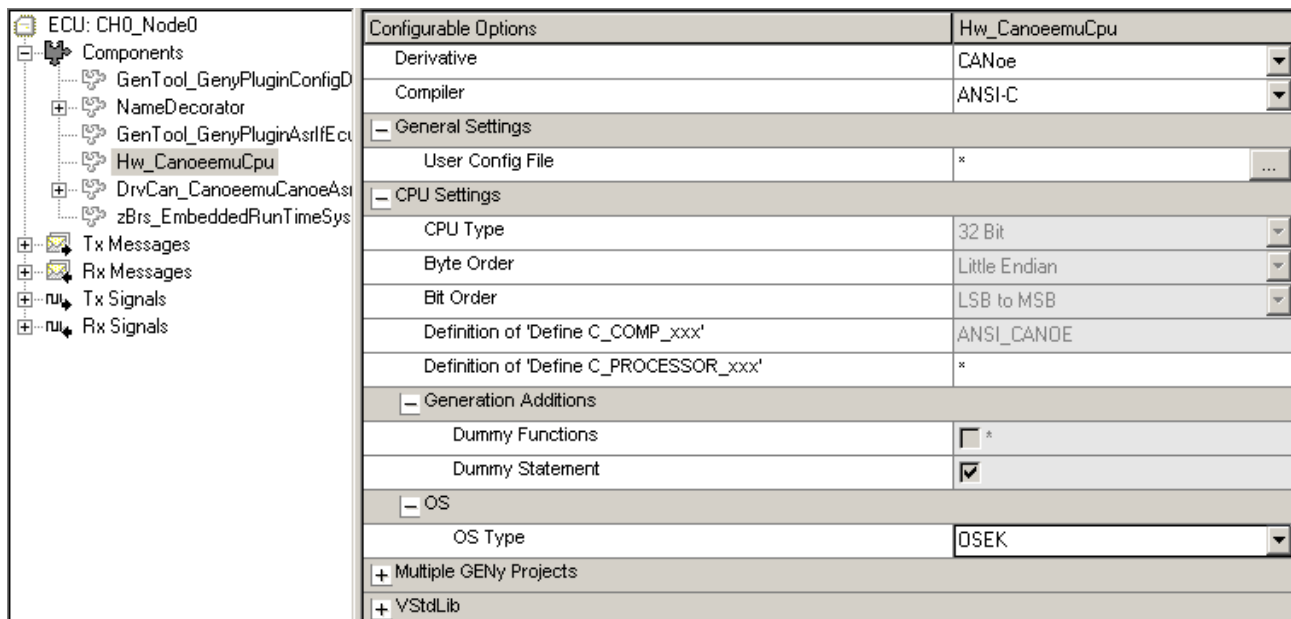


Figure 6-1 Select OS Type

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

- Category 2 (only for OSEK OS or AUTOSAR OS):
A macro "ISR(<ISR Names>_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).
switch: C_DISABLE_OSEK_OS_INTCAT2
- Void-Void Interrupt Function:
Like in Category 1 the Interrupt is not handled by OS and the ISR is declared as void ISR(void) and has to be called by interrupt controller in case of an CAN interrupt.
switch: C_ENABLE_ISRVOID

6.1.5 ISR Names

Prototype	
void CanIsrRx_x (void) → where x is the hardware controller index	
Parameter	
---	---
Return code	
---	---
Functional Description	
ISR handle the message reception indication.	
Particularities and Limitations	
■ -	

Prototype	
void CanIsrTx_x (void) → where x is the hardware controller index	
Parameter	
---	---
Return code	
---	---
Functional Description	
ISR handle the message transmission confirmation.	
Particularities and Limitations	
■ -	


Prototype
void CanIsrErr_x (void) → where x is the hardware controller index

Parameter	
---	---
Return code	
---	---
Functional Description	
ISR handle the bus off notification.	
Particularities and Limitations	
<div> <div></div> <div>-</div> </div>	

Prototype	
void CanIsrWakeup_x (void) → where x is the hardware controller index	
Parameter	
---	---
Return code	
---	---
Functional Description	
ISR handle the wakeup notification.	
Particularities and Limitations	
<div> <div></div> <div>-</div> </div>	

6.2 Services provided by CAN

6.2.1 Can_InitMemory

Prototype	
void Can_InitMemory (void)	
Parameter	
-	
Return code	
void	-
Functional Description	
<p>Service initializes module global variables, which cannot be initialized in the startup code.</p> <p>Use this to re-run the system without performing a new start from power on.</p> <p>(E.g.: used to support an ongoing debug session without a complete re-initialization.)</p> <p>Must be followed by a call to "Can_Init()".</p>	
Particularities and Limitations	
Called by Application.	
	<p>Caution</p> <p>None AUTOSAR API</p>

Call Context

Should be called while power on initialization before "Can_Init()" on task level.

- This function is Synchronous
- This function is Non-Reentrant
- Availability: Always

Table 6-1 Can_InitMemory

6.2.2 Can_Init

Prototype

```
void Can_Init (Can_ConfigPtrType ConfigPtr)
```

Parameter

ConfigPtr	<p>Pointer to the configuration data structure.</p> <p>When using the "Multiple ECU" configuration feature, then for each Identity the appropriate "CanConfig_<Identity>"-structure exists and has to be chosen here.</p>
-----------	---

Return code

void	-
------	---

Functional Description

This function initializes global CAN driver variables during ECU start-up.

Particularities and Limitations

Called by CanInterface.

Call Context

- Has to be called during start-up before CAN communication.
 - Must be called before calling "Can_InitController()" but after call of "Can_InitMemory()".
- Parameter "ConfigPtr" will be taken into account only for "Multiple ECU Configurataion" and in Post-Build variant.
- Disabled Interrupts.
- This function is Synchronous
 - This function is Non-Reentrant
 - Availability: Always

Table 6-2 Can_Init

6.2.3 Can_InitController

Prototype

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

Parameter

Controller	Number of controller
Config	Pointer to baud rate configuration structure

Return code	
void	-
Functional Description	
<p>Initialization of controller specific CAN hardware.</p> <p>The CAN driver registers and variables are initialized.</p> <p>The CAN controller is fully initialized and left back within the state "Stop Mode", ready to change to "Running Mode".</p>	
Particularities and Limitations	
Called by CanInterface.	
Call Context	
<ul style="list-style-type: none"> ■ Has to be called during the startup sequence before CAN communication takes place but after calling "Can_Init()". ■ Must not be called while in "Sleep Mode". <p>Disabled Interrupts.</p> <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: MICROSAR401 only 	

Table 6-3 Can_InitController

6.2.4 Can_InitController

Prototype	
<pre>void Can_InitController (uint8 Controller, Can_ControllerConfigPtrType ControllerConfigPtr)</pre>	
Parameter	
Controller	Number of controller
Config	Pointer to the configuration data structure.
Return code	
void	-
Functional Description	
<p>Initialization of controller specific CAN hardware.</p> <p>The CAN driver registers and variables are initialized.</p> <p>The CAN controller is fully initialized and left back within the state "stop mode", ready to change to "Running Mode".</p>	
Particularities and Limitations	
Called by CanInterface.	
Call Context	

- Has to be called during the startup sequence before CAN communication takes place but after calling "Can_Init()".
- Must not be called while in "Sleep Mode".

Disabled Interrupts

- This function is Synchronous
- This function is Non-Reentrant
- Availability: MICROSAR3 only

Table 6-4 Can_InitController

6.2.5 Can_ChangeBaudrate

Prototype	
Std_ReturnType Can_ChangeBaudrate (uint8 Controller, const uint16 Baudrate)	
Parameter	
Controller	Number of controller to be changed
Baudrate	Baud rate to be set
Return code	
Std_ReturnType	<ul style="list-style-type: none"> ■ E_NOT_OK Baud rate is not set ■ E_OK Baud rate is set
Functional Description	
This service shall change the baud rate and reinitialize the CAN controller.	
Particularities and Limitations	
Called by Application.	
Call Context	
<p>Has to be called during the startup sequence before CAN communication takes place but after calling "Can_Init()".</p> <p>The CAN controller must be in "Stop Mode".</p> <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: MICROSAR403 only & if "CanChangeBaudrateApi" is activated or "CanSetBaudrateApi" is de-activated. 	

Table 6-5 Can_ChangeBaudrate

6.2.6 Can_CheckBaudrate

Prototype	
Std_ReturnType Can_CheckBaudrate (uint8 Controller, const uint16 Baudrate)	
Parameter	
Controller	Number of controller to be checked

Baudrate	Baud rate to be checked
Return code	
Std_ReturnType	<ul style="list-style-type: none"> ■ E_NOT_OK Baud rate is not available ■ E_OK Baud rate is available
Functional Description	
This service shall check if the given baud rate is supported of the CAN controller.	
Particularities and Limitations	
Called by Application.	
Call Context	
<ul style="list-style-type: none"> ■ Must not be called nested. ■ Only available if "CanChangeBaudrateApi" is activated. <p>The CAN controller must be initialized.</p> <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: MICROSAR403 only & "CanChangeBaudrateApi" is activated ("CAN_CHANGE_BAUDRATE_SUPPORT == STD_ON") 	

Table 6-6 Can_CheckBaudrate

6.2.7 Can_SetBaudrate

Prototype	
Std_ReturnType Can_SetBaudrate (uint8 Controller, uint16 BaudRateConfigID)	
Parameter	
Controller	Number of controller to be set
BaudRateConfigID	Identity of the configured baud rate (available as Symbolic Name)
Return code	
Std_ReturnType	<ul style="list-style-type: none"> ■ E_NOT_OK Baud rate is not set ■ E_OK Baud rate is set
Functional Description	
<p>This service shall change the baud rate and reinitialize the CAN controller.</p> <p>(Similar to "Can_ChangeBaudrate()" but used when identical baud rates are used for different CAN FD settings).</p>	
Particularities and Limitations	
Called by Application.	
Call Context	
<ul style="list-style-type: none"> ■ Must not be called nested. ■ Only available if "CanSetBaudrateApi" is activated. ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: MICROSAR403 only & "CanSetBaudrateApi" is activated ("CAN_SET_BAUDRATE_API == STD_ON") 	

Table 6-7 Can_SetBaudrate

6.2.8 Can_InitStruct


Prototype	
void Can_InitStruct (uint8 Controller, uint8 Index)	
Parameter	
Controller	Number of the controller to be changed
Index	Index of the initialization structure to be used for baud rate and mask settings
Return code	
void	-
Functional Description	
<p>Set content of the initialization structure (before calling "Can_InitController()").</p> <p>Service function to change the initialization structure setup left behind by the Generation Tool.</p> <p>The structure contains information about baud rate and filter settings.</p> <p>Subsequent "Can_InitController()" must be called to activate these settings.</p>	
Particularities and Limitations	
Called by Application.	
	<p>Caution</p> <p>None AUTOSAR API</p>
Call Context	
<p>Call this function between calling "Can_Init()" and "Can_InitController()".</p> <p>"Can_Init" was called.</p> <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: MICROSAR3 only 	

Table 6-8 Can_InitStruct

6.2.9 Can_GetVersionInfo

Prototype	
void Can_GetVersionInfo (Can_VersionInfoPtrType VersionInfo)	

Parameter	
VersionInfo	Pointer to where to store the version information of the CAN driver. <pre>typedef struct { uint16 vendorID; uint16 moduleID; MICROSAR3 only: uint8 instanceID; uint8 sw_major_version; (MICROSAR3 only: BCD coded) uint8 sw_minor_version; (MICROSAR3 only: BCD coded) uint8 sw_patch_version; (MICROSAR3 only: BCD coded) } Std_VersionInfoType;</pre>
Return code	
void	-
Functional Description	
Get the version information of the CAN driver.	
Particularities and Limitations	
Called by Application.	
Call Context	
Only available if "CanVersionInfoApi" is activated. <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Reentrant ■ Availability: "CanVersionInfoApi" is activated ("CAN_VERSION_INFO_API == STD_ON") 	

Table 6-9 Can_GetVersionInfo

6.2.10 Can_GetStatus

Prototype	
uint8 Can_GetStatus (uint8 Controller)	
Parameter	
Controller	Number of the controller requested for status information
Return code	
uint8	<ul style="list-style-type: none"> ■ CAN_STATUS_STOP (Bit coded status information) ■ CAN_STATUS_INIT ■ CAN_STATUS_INCONSISTENT, CAN_DEACTIVATE_CONTROLLER (only with "CanRamCheck" active) ■ CAN_STATUS_WARNING ■ CAN_STATUS_PASSIVE ■ CAN_STATUS_BUSOFF ■ CAN_STATUS_SLEEP

Functional Description

Delivers the status of the hardware.

Only one of the status bits CAN_STATUS_SLEEP/STOP/BUSOFF/PASSIVE/WARNING is set.

The CAN_STATUS_INIT bit is always set if a controller is initialized.

CAN_STATUS_SLEEP has the highest and CAN_STATUS_WARNING the lowest priority.

CAN_STATUS_INCONSISTENT will be set if one Common CAN channel. is not "Stop" or "Sleep".

CAN_DEACTIVATE_CONTROLLER is set in case the "CanRamCheck" detected an Issue.

"status" can be analyzed using the provided API macros:

CAN_HW_IS_OK(status): return "true" in case no warning, passive or bus off occurred.

CAN_HW_IS_WARNING(status): return "true" in case of waning status.

CAN_HW_IS_PASSIVE(status): return "true" in case of passive status.

CAN_HW_IS_BUSOFF(status): return "true" in case of bus off status (may be already false in Notification).

CAN_HW_IS_WAKEUP(status): return "true" in case of not in sleep mode.

CAN_HW_IS_SLEEP(status): return "true" in case of sleep mode.

CAN_HW_IS_STOP(status): return "true" in case of stop mode.

CAN_HW_IS_START(status): return "true" in case of not in stop mode.

CAN_HW_IS_INCONSISTENT(status): return "true" in case of an inconsistency between two common CAN channels.

Particularities and Limitations

Called by network management or Application.



Caution

None AUTOSAR API

Call Context

- This function is Synchronous
- This function is Non-Reentrant
- Availability: "CanGetStatus" is activated ("CAN_GET_STATUS == STD_ON")

Table 6-10 Can_GetStatus

6.2.11 Can_SetControllerMode

Prototype

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

Parameter

Controller	Number of the controller to be set
Transition	Requested transition to destination mode

Return code

Can_ReturnType	<ul style="list-style-type: none"> ■ CAN_NOT_OK mode change unsuccessful ■ CAN_OK mode change successful
----------------	--

Functional Description
Change the controller mode to the following possible destination values: CAN_T_START, CAN_T_STOP, CAN_T_SLEEP, CAN_T_WAKEUP.
Particularities and Limitations
Called by CanInterface.
Call Context
Must not be called within CAN driver context like RX, TX or Bus Off callouts. Interrupts locked by CanInterface
<ul style="list-style-type: none"> ■ This function is MICROSAR3: Synchronous / MICROSAR4: Asynchronous ■ This function is Non-Reentrant ■ Availability: Always

Table 6-11 Can_SetControllerMode

6.2.12 Can_ResetBusOffStart


Prototype	
void Can_ResetBusOffStart (uint8 Controller)	
Parameter	
Controller	Number of the controller
Return code	
void	-
Functional Description	
This is a compatibility function (for a CANbedded protocol stack) used during the start of the Bus Off handling to remove the Bus Off state.	
Particularities and Limitations	
Called by CAN driver.	
	Caution None AUTOSAR API
Call Context	
Called while BusOff event handling (Polling or Interrupt context).	
<ul style="list-style-type: none">■ This function is Synchronous■ This function is Non-Reentrant■ Availability: Always	

Table 6-12 Can_ResetBusOffStart

6.2.13 Can_ResetBusOffEnd


Prototype	
void Can_ResetBusOffEnd (uint8 Controller)	
Parameter	
Controller	Number of the controller
Return code	
void	-
Functional Description	
This is a compatibility function (for a CANbedded protocol stack) used during the end of the Bus Off handling to remove the Bus Off state.	
Particularities and Limitations	
Called by CAN driver.	
	Caution None AUTOSAR API
Call Context	
Called inside "Can_SetControllerMode()" while Start transition. <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: Always 	

Table 6-13 Can_ResetBusOffEnd

6.2.14 Can_Write

Prototype	
Can_ReturnType Can_Write (Can_HwHandleType Hth, Can_PduInfoPtrType PduInfo)	
Parameter	
Hth	Handle of the mailbox intended to send the message
PduInfo	Information about the outgoing message (ID, DLC, data)
Return code	
Can_ReturnType	<ul style="list-style-type: none"> ■ CAN_NOT_OK transmit unsuccessful ■ CAN_OK transmit successful ■ CAN_BUSY transmit could not be accomplished due to the controller is busy.
Functional Description	
Send a CAN message over CAN.	
Particularities and Limitations	
Called by CanInterface.	
Call Context	

- Called by the CanInterface with at least disabled CAN interrupts.
- (Due to data security reasons the CanInterface has to accomplish this and thus it is not needed a further more in the CAN Driver.)

CAN Interrupt locked.

- This function is Synchronous
- This function is Non-Reentrant
- Availability: Always

Table 6-14 Can_Write

6.2.15 Can_CancelTx


Prototype	
void Can_CancelTx (Can_HwHandleType Hth, PduIdType PduId)	
Parameter	
Hth	Handle of the mailbox intended to be cancelled.
PduId	Pdu identifier
Return code	
void	-
Functional Description	
Cancel the TX message in the hardware buffer (if possible) or mark the message as not to be confirmed in case of the cancellation is unsuccessful.	
Particularities and Limitations	
Called by CanTp or Application.	
	Caution None AUTOSAR API
Call Context	
Called by CanTp or Application. <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: Always 	

Table 6-15 Can_CancelTx

6.2.16 Can_SetMirrorMode

Prototype	
void Can_SetMirrorMode (uint8 Controller, CddMirror_MirrorModeType mirrorMode)	
Parameter	
Controller	Number of the controller to be mirror'ed.
mirrorMode	


Return code	
void	-
Functional Description	
Switch the Appl_GenericPreCopy/Confirmation function ON or OFF.	
Particularities and Limitations	
Called by "Mirror Mode" CDD.	
	Caution None AUTOSAR API
Call Context	
<ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: "C_ENABLE_MIRROR_MODE" is defined (user configuration file) 	

Table 6-16 Can_SetMirrorMode

6.2.17 Can_CheckWakeup

Prototype	
Std_ReturnType Can_CheckWakeup (uint8 Controller)	
Parameter	
Controller	Number of the controller to be checked for Wake Up events.
Return code	
Std_ReturnType	<ul style="list-style-type: none"> ■ E_OK the given controller caused a Wake Up before. ■ E_NOT_OK the given controller caused no Wake Up before.
Functional Description	
Service function to check the occurrence of Wake Up events for the given controller (used as Wake Up callback for higher layers).	
Particularities and Limitations	
Called by CanInterface.	
Call Context	
Called while Wakeup validation phase. <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: In AR4.x named "Can_CheckWakeup", in AR3.x named "Can_Cbk_CheckWakeup" (Name mapped by define) 	

Table 6-17 Can_CheckWakeup

6.2.18 Can_DisableControllerInterrupts

Prototype	
<code>void Can_DisableControllerInterrupts (uint8 Controller)</code>	
Parameter	
Controller	Number of the CAN controller to disable interrupts for.
Return code	
void	-
Functional Description	
Service function to disable the CAN interrupt for the given controller (e.g. due to data security reasons).	
Particularities and Limitations	
Called by SchM.	
Call Context	
<p>Called within Critical Area handling or out of Application code.</p> <p>Must not be called while CAN controller is in sleep mode.</p> <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: Always 	

Table 6-18 Can_DisableControllerInterrupts

6.2.19 Can_EnableControllerInterrupts

Prototype	
<code>void Can_EnableControllerInterrupts (uint8 Controller)</code>	
Parameter	
Controller	Number of the CAN controller to disable interrupts for.
Return code	
void	-
Functional Description	
Service function to (re-)enable the CAN interrupt for the given controller (e.g. due to data security reasons).	
Particularities and Limitations	
Called by SchM.	
Call Context	
<p>Called within Critical Area handling or out of Application code.</p> <p>Must not be called while CAN controller is in sleep mode.</p> <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: Always 	

Table 6-19 Can_EnableControllerInterrupts

6.2.20 Can_MainFunction_Write

Prototype	
<code>void Can_MainFunction_Write (void)</code>	
Parameter	
-	
Return code	
void	-
Functional Description	
Service function to poll TX events (confirmation, cancellation) for all controllers and all TX mailboxes to accomplish the TX confirmation handling (like CanInterface notification).	
Particularities and Limitations	
Called by SchM.	
Call Context	
<p>Called within cyclic TX task.</p> <p>Must not interrupt the call of "Can_Write()".</p> <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: Always 	

Table 6-20 Can_MainFunction_Write

6.2.21 Can_MainFunction_Read

Prototype	
<code>void Can_MainFunction_Read (void)</code>	
Parameter	
-	
Return code	
void	-
Functional Description	
<p>Service function to poll RX events for all controllers and all RX mailboxes to accomplish the RX indication handling (like CanInterface notification).</p> <p>Also used for a delayed read (from task level) of the RX Queue messages which were queued from interrupt context.</p>	
Particularities and Limitations	
Called by SchM.	
Call Context	
<p>Called within cyclic RX task.</p> <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: Always 	

Table 6-21 Can_MainFunction_Read

6.2.22 Can_MainFunction_BusOff

Prototype	
void Can_MainFunction_BusOff (void)	
Parameter	
-	
Return code	
void	-
Functional Description	
Polling of Bus Off events to accomplish the Bus Off handling. Service function to poll Bus Off events for all controllers to accomplish the Bus Off handling (like calling of "CanIf_ControllerBusOff()" in case of Bus Off occurrence).	
Particularities and Limitations	
Called by SchM.	
Call Context	
Called within cyclic BusOff task.	
<ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: Always 	

Table 6-22 Can_MainFunction_BusOff

6.2.23 Can_MainFunction_Wakeup

Prototype	
void Can_MainFunction_Wakeup (void)	
Parameter	
-	
Return code	
void	-
Functional Description	
Service function to poll Wake Up events for all controllers to accomplish the Wake Up handling (like calling of "CanIf_SetWakeupEvent()" in case of Wake Up occurrence).	
Particularities and Limitations	
Called by SchM.	
Call Context	
Called within cyclic Wakeup task.	
<ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: Always 	

Table 6-23 Can_MainFunction_Wakeup

6.2.24 Can_MainFunction_Mode


Prototype	
<pre>void Can_MainFunction_Mode (void) Can_ReturnType Appl_GenericPreCopy(uint8 Controller</pre>	
Parameter	
-	
Controller	Controller which received the message
ID	ID of the received message. In case of extended or mixed ID systems the highest bit (bit 31) is set to mark an extended ID.
DLC	DLC of the received message.
pData	Pointer to the data of the received message.
Return code	
void	-
Functional Description	
Service function to poll Mode changes over all controllers. (This is handled asynchronous if not accomplished in "Can_SetControllerMode()").	
Particularities and Limitations	
Called by SchM.	
	Caution Application callback function which informs about all incoming RX messages including the contained data
Call Context	
Called within cyclic mode change task. <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: MICROSAR4x only 	

Table 6-24 Can_MainFunction_Mode

6.2.25 Appl_GenericConfirmation

Prototype	
<pre>Can_ReturnType Appl_GenericConfirmation (PduIdType PduId)</pre>	
Parameter	
PduId	Handle of the PDU specifying the message.
Return code	
Can_ReturnType	<ul style="list-style-type: none"> ■ CAN_OK Higher layer (CanInterface) confirmation will be called. ■ CAN_NOT_OK No further higher layer (CanInterface) confirmation will be called.


Functional Description	
Application callback function which informs about TX messages being sent to the CAN bus.	
Particularities and Limitations	
Called by CAN driver.	
	Caution None AUTOSAR API
Call Context	
Called within CAN message transmission finished context (Polling or Interrupt). "PduId" is read only and must not be accessed for further write operations.	
<ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: "CanGenericConfirmation" is activated ("CAN_GENERIC_CONFIRMATION == STD_ON") & "CanIfTransmitBuffer" activated (in CanInterface). 	

Table 6-25 Appl_GenericConfirmation

6.2.26 Appl_GenericConfirmation


Prototype	
Can_ReturnType Appl_GenericConfirmation (uint8 Controller, Can_PduInfoPtrType DataPtr)	
Parameter	
Controller	Number of the causing controller.
DataPtr	
Return code	
Can_ReturnType	CAN_OK Higher layer (CanInterface) confirmation will be called. CAN_NOT_OK No further higher layer (CanInterface) confirmation will be called.
Functional Description	
Application callback function which informs about TX messages being sent to the CAN bus.	
Particularities and Limitations	
Called by CAN driver.	
	Caution None AUTOSAR API
Call Context	
Called within CAN message transmission finished context (Polling or Interrupt). If "Generic Confirmation" and "Transmit Buffer" (both set in CanInterface) are active, then the switch "Cancel Support Api" is also needed (also set in CanIf), otherwise a compiler error occurs.	
<ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: "CanGenericConfirmation" is set to API2 ("CAN_GENERIC_CONFIRMATION == CAN_API2"). 	

Table 6-26 Appl_GenericConfirmation

6.2.27 Appl_GenericPreTransmit


Prototype	
void Appl_GenericPreTransmit (uint8 Controller, Can_PduInfoPtrType_var DataPtr)	
Parameter	
Controller	Number of the controller on which the hardware observation takes place.
DataPtr	Pointer to a Can_PduType structure including ID, DLC, Pdu and data pointer.
Return code	
void	-
Functional Description	
Application callback function allowing the modification of the data to be transmitted (e.g.: add CRC).	
Particularities and Limitations	
Called by CAN driver.	
	Caution None AUTOSAR API
Call Context	
Called within "Can_Write()". <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: "CanGenericPretransmit" is activated ("CAN_GENERIC_PRETRANSMIT == STD_ON"). 	

Table 6-27 Appl_GenericPreTransmit

6.2.28 ApplCanTimerStart

Prototype	
void ApplCanTimerStart (CanChannelHandle Controller, uint8 source)	
Parameter	
Controller	Number of the controller on which the hardware observation takes place. (only if not using "Optimize for one controller")
source	Source for the hardware observation (see chapter Hardware Loop Check / Timeout Monitoring).
Return code	
void	-
Functional Description	
Service function to start an observation timer (see chapter Hardware Loop Check / Timeout Monitoring).	
Particularities and Limitations	
Called by CAN driver.	

**Caution**

None AUTOSAR API

Call Context

For context information please refer to chapter "Hardware Loop Check".

- This function is Synchronous
- This function is Non-Reentrant
- Availability: "CanHardwareCancelByAppl" is activated ("CAN_HW_LOOP_SUPPORT_API == STD_ON").

Table 6-28 ApplCanTimerStart

6.2.29 ApplCanTimerLoop

Prototype

Can_ReturnType **ApplCanTimerLoop** (CanChannelHandle Controller, uint8 source)

Parameter

Controller	Number of the controller on which the hardware observation takes place. (only if not using "Optimize for one controller")
source	Source for the hardware observation (see chapter Hardware Loop Check / Timeout Monitoring).

Return code

Can_ReturnType	<ul style="list-style-type: none"> ■ CAN_NOT_OK when loop shall be broken (observation stops) ■ CAN_NOT_OK should only be used in case of a timeout occurs due to a hardware issue. ■ After this an appropriate error handling is needed (see chapter Hardware Loop Check / Timeout Monitoring). ■ CAN_OK when loop shall be continued (observation continues)
----------------	--

Functional Description

Service function to check (against generated max loop value) whether a hardware loop shall be continued or broken.

Particularities and Limitations

Called by CAN driver.

**Caution**

None AUTOSAR API

Call Context

For context information please refer to chapter "Hardware Loop Check".

- This function is Synchronous
- This function is Non-Reentrant
- Availability: "CanHardwareCancelByAppl" is activated ("CAN_HW_LOOP_SUPPORT_API == STD_ON").

Table 6-29 ApplCanTimerLoop

6.2.30 ApplCanTimerEnd


Prototype	
void ApplCanTimerEnd (CanChannelHandle Controller, uint8 source)	
Parameter	
Controller	Number of the controller on which the hardware observation takes place. (only if not using "Optimize for one controller")
source	Source for the hardware observation (see chapter Hardware Loop Check / Timeout Monitoring).
Return code	
void	-
Functional Description	
Service function to to end an observation timer (see chapter Hardware Loop Check / Timeout Monitoring).	
Particularities and Limitations	
Called by CAN driver.	
	Caution None AUTOSAR API
Call Context	
For context information please refer to chapter "Hardware Loop Check". <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: "CanHardwareCancelByAppl" is activated ("CAN_HW_LOOP_SUPPORT_API == STD_ON"). 	

Table 6-30 ApplCanTimerEnd

6.2.31 ApplCanInterruptDisable

Prototype	
void ApplCanInterruptDisable (uint8 Controller)	
Parameter	
Controller	Number of the controller for the CAN interrupt lock.
Return code	
void	-
Functional Description	
Service function to support the disabling of CAN Interrupts by the application. E.g.: the CAN driver itself should not access the common Interrupt Controller due to application specific restrictions (like security level etc.). Or the application like to be informed because of an CAN interrupt lock.	
Particularities and Limitations	
Called by CAN driver.	


	Caution None AUTOSAR API
Call Context	
<p>Called by the CAN Driver within "Can_DisableControllerInterrupts()".</p> <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: "CanInterruptLock" is set to APPL or BOTH ("CAN_INTLOCK == CAN_APPL" or "CAN_INTLOCK == CAN_BOTH"). 	

Table 6-31 ApplCanInterruptDisable

6.2.32 ApplCanInterruptRestore


Prototype	
void ApplCanInterruptRestore (uint8 Controller)	
Parameter	
Controller	Number of the controller for the CAN interrupt unlock.
Return code	
void	-
Functional Description	
<p>Service function to support the enabling of CAN Interrupts by the application.</p> <p>E.g.: the CAN driver itself should not access the common Interrupt Controller due to application specific restrictions (like security level etc.). Or the application like to be informed because of an CAN interrupt lock.</p>	
Particularities and Limitations	
Called by CAN driver.	
	Caution None AUTOSAR API
Call Context	
<p>Called by the CAN Driver within "Can_EnableControllerInterrupts()".</p> <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: "CanInterruptLock" is set to APPL or BOTH ("CAN_INTLOCK == CAN_APPL" or "CAN_INTLOCK == CAN_BOTH"). 	

Table 6-32 ApplCanInterruptRestore

6.2.33 Appl_CanOverrun

Prototype	
void Appl_CanOverrun (uint8 Controller)	


Parameter	
Controller	Number of the controller for which the overrun was detected.
Return code	
void	-
Functional Description	
<p>This function will be called when an overrun is detected for a BasicCAN mailbox.</p> <p>Alternatively a DET call can be selected instead of ("CanOverrunNotification" is set to "DET").</p>	
Particularities and Limitations	
Called by CAN driver.	
	<p>Caution</p> <p>None AUTOSAR API</p>
Call Context	
<p>Called within CAN message reception or error detection context (Polling or Interrupt).</p> <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: "CanOverrunNotification" set to APPL ("CAN_OVERRUN_NOTIFICATION == CAN_APPL"). 	

Table 6-33 Appl_CanOverrun

6.2.34 Appl_CanFullCanOverrun


Prototype	
void Appl_CanFullCanOverrun (uint8 Controller)	
Parameter	
Controller	Number of the controller for which the overrun was detected.
Return code	
void	-
Functional Description	
<p>This function will be called when an overrun is detected for a FullCAN mailbox.</p> <p>Alternatively a DET call can be selected instead of ("CanOverrunNotification" is set to "DET").</p>	
Particularities and Limitations	
Called by CAN driver.	
	<p>Caution</p> <p>None AUTOSAR API</p>
Call Context	
<p>Called within CAN message reception or error detection context (Polling or Interrupt).</p> <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: "CanOverrunNotification" set to APPL ("CAN_OVERRUN_NOTIFICATION == CAN_APPL"). 	

Table 6-34 Appl_CanFullCanOverrun

6.2.35 Appl_CanCorruptMailbox


Prototype	
void Appl_CanCorruptMailbox (uint8 Controller, Can_HwHandleType hwObjHandle)	
Parameter	
Controller	Number of the controller for which the check failed.
hwObjHandle	Hardware handle of the defect mailbox.
Return code	
void	-
Functional Description	
This function will notify the application (during "Can_InitController()") about a defect mailbox within the CAN cell.	
Particularities and Limitations	
Called by CAN driver.	
	Caution None AUTOSAR API
Call Context	
Call within controller initialization. <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: "CanRamCheck" set to "MailboxNotifiatiion" ("CAN_RAM_CHECK == CAN_NOTIFY_MAILBOX"). 	

Table 6-35 Appl_CanCorruptMailbox

6.2.36 Appl_CanRamCheckFailed

Prototype	
uint8 Appl_CanRamCheckFailed (uint8 Controller)	
Parameter	
Controller	Number of the controller for which the check failed
Return code	
uint8	<ul style="list-style-type: none"> ■ action With this "action" the application can decide how to proceed with the initialization. ■ CAN_DEACTIVATE_CONTROLLER - deactivate the controller ■ CAN_ACTIVATE_CONTROLLER - activate the controller
Functional Description	
This function will notify the application (during "Can_InitController()") about a defect CAN controller due to a previous failed mailbox check.	


Particularities and Limitations	
Called by CAN driver.	
	Caution None AUTOSAR API
Call Context	
Call within controller initialization. <ul style="list-style-type: none"> ■ This function is Synchronous ■ This function is Non-Reentrant ■ Availability: "CanRamCheck" set to "Active" or "MailboxNotifiation" ("CAN_RAM_CHECK != CAN_NONE"). 	

Table 6-36 Appl_CanRamCheckFailed

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

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	<p>CanIf_CancelTxNotification (non AUTOSAR) A special Software cancellation callback only used within Vector CAN driver CAN Interface bundle.</p> <p>CanIf_TxConfirmation Notification for a successful transmission. (see [4])</p> <p>CanIf_CancelTxConfirmation Notification for a successful TX cancellation. (see [4])</p> <p>CanIf_RxIndication Notification for a message reception. (see [4])</p> <p>CanIf_ControllerBusOff Bus Off notification function. (see [4])</p> <p>CanIf_ControllerModeIndication MICROSAR4x only: Notification for mode successfully changed.</p>
Os (MICROSAR4x)	<p>OS_TICKS2MS_<counterShortName>() OS macro to get timebased ticks from counter.</p> <p>GetElapsedValue Get elapsed tick count.</p> <p>GetCounterValue Get tick count start.</p>

Table 6-37 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_MSR403/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)
- Rx Queue size
- Controller mapping (mapping of logical channel to hardware node).
- Controller specific Wakeup support.

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 da DaVinci Configurator

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

8 AUTOSAR Standard Compliance

8.1 Limitations / Restrictions

Category	Description	Version
Functional	No multiple AUTOSAR CAN driver allowed in the system	3.0.6
Functional	No support for L-PDU callout (AUTOSAR 3.2.1), but support 'Generic Precopy' instead	3.2.1
Functional	No support for multiple read and write period configuration	3.2.1
API	"Symbolic Name Values" may change their values after precompile phase so do not use it for Linktime or Postbuild variants. It's recommended that higher layer generator use Values (ObjectIDs) from EcuC file. Vector CAN Interface does so.	3.0.6

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