

MICROSAR CAN Driver

Technical Reference

Renesas

RH850/P1x-C

MCAN

Version 1.02.00

Authors	Cengiz Ünver, Peter Herrmann
Status	Released

Document Information

History Core

Author	Date	Version	Remarks
Holger Birke	2006-06-21	1.0	Initial version
Holger Birke	2006-06-28	1.1	Review modifications
Holger Birke	2006-10-26	1.2	New feature Tx polling, FullCAN Tx and support DEM
Holger Birke	2007-01-22	1.3	New feature Bus Off Polling
Holger Birke	2007-02-15	1.4	Minor Changes
Holger Birke	2007-07-10	1.5	ASR2.1
Holger Birke	2007-08-24	1.6	Renaming MICROSAR
Holger Birke	2007-08-28	1.7	Remove Driver version
Holger Birke	2007-08-29	1.8	Driver version also removed from Chapter 3
Holger Birke	2007-11-13	1.9	Changed API Can_Init(), add API Can_InitStruct(), add init structure description (HL2.22)
Holger Birke	2007-12-03	1.10	Improve Interrupt description
Holger Birke	2008-02-20	1.11	ASR3
Holger Birke	2008-04-18	1.12	Review Reworks (Sh2 review and by visem)
Holger Birke	2008-07-21	1.13	Review Reworks (TMS320)
Holger Birke	2008-08-13	1.14	Core 3.3 Optimization for runtime, ROM and RAM
Holger Birke	2008-08-13	1.15	Core 3.5 rename INTERRUPT & POLLING Update Tool configuration description Add Remote Frame rejection description
Holger Birke	2008-10-23	1.16	Core 3.6 add new API handle "Hardware Loop Check" by application + beautifying
Holger Birke	2009-02-06	1.17	Core 3.7 Add individual polling
Holger Birke	2009-05-19	1.18	Improve "Generic Precopy" description (extended ID bit) Add Compiler and Memory abstraction, Add possibility to report CAN_E_TIMEOUT as DET.
Holger Birke	2009-07-15	1.18.01	Core 3.09 Remove Compiler abstraction CAN_ISR. Change "Hardware Loop Check" naming.

Holger Birke	2009-07-28	1.18.02	Review reworks
Holger Birke	2009-10-01	1.18.03	Core 3.10 Add RxQueue (high-end) and Generic Confirmation
Holger Birke	2010-02-04	1.19	Core 3.11 Add "Multiple BasicCAN", "Support Mixed ID", "Optimize for one controller", "Dynamic FullCAN Tx ID" and "Size of Hw HandleType". Rename "Hardware Cancellation" Correct "Services used by CAN"
Holger Birke	2010-04-01	1.20	Core 3.12 Add Critical Section description Add "Common CAN" Add Hardware assertion (DET) description Add Can_GetStatus() + Interrupt category configuration. Add ApplCanInterruptDisable/Restore()
Holger Birke	2010-11-24	2.00	Core 4.00 Update to MICROSAR4 Add "Overrun notification" Add "RAM check"
Holger Birke	2011-04-18	2.00.01	Review reworks (VJ)
Holger Birke	2011-06-28	2.00.02	Rework (add missing config settings to GENy GUI description) Add MicroSar – AUTOSAR deviations
Holger Birke	2011-07-29	2.01	Core 4.01 Add "GenericPreTransmit"
Holger Birke	2012-01-13	2.01.01	Improve description for "Nested Interrupts" and "Identical ID cancellation"
Holger Birke	2012-04-02	2.02.00	Core 4.02 Add Platform, CANCell and Manufacturer as First Page Information Add Void-Void ISR configuration, support ASR3.2.1 Identical ID cancellation
Holger Birke	2012-04-02	2.03.00	Partial Network part of configuration (no more preconfig)
Holger Birke	2012-06-29	2.04.00	Core 4.03 Support AR4-R5 (ASR4.0.3) – New API added Improve Hardware Loop description
Holger Birke	2012-11-07	2.05.00	Core 4.04 Add Re-initialization description Instance ID of DET is always 0
Holger Birke	2012-11-07	2.05.01	Improve Hardware Loop description

Holger Birke	2013-10-11	2.06.00	Add CAN FD description (Can_SetBaudrate() API)
--------------	------------	---------	--

History Platforms

Author	Date	Version	Remarks
C. Ünver	2015-04-27	1.00.00	Initial version
P. Herrmann	2016-01-28	1.01.00	Added MCAN Rev. 3.1.0 changes. Additional description concerning the Bosch MCAN Errata Sheet.
P. Herrmann	2016-10-06	1.02.00	Additional description concerning the Bosch MCAN Errata Sheet.

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
[6]	M_CAN Controller Area Network Errata Sheet	REL2015 0701

1.1 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.



Caution

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

1.1	Scope of the Document.....	4
2	Hardware Overview	8
3	Introduction.....	9
3.1	Architecture Overview	10
4	Functional Description	12
4.1	Features	12
4.2	Initialization	15
4.3	Communication	16
4.4	States / Modes	18
4.5	Re-Initialization	19
4.6	CAN Interrupt Locking.....	19
4.7	Main Functions	19
4.8	Error Handling.....	20
4.9	Common CAN.....	24
5	Integration.....	27
5.1	Scope of Delivery.....	27
5.2	Include Structure.....	28
5.3	Critical Sections	28
5.4	Compiler Abstraction and Memory Mapping.....	30
6	Hardware Specific Hints.....	32
7	API Description.....	35
7.1	Interrupt Service Routines provided by CAN	35
7.2	Services provided by CAN	36
7.3	Services used by CAN	60
8	Configuration.....	62
8.1	Pre-Compile Parameters.....	62
8.2	Link-Time Parameters	63
8.3	Post-Build Parameters	63
8.4	Configuration with da DaVinci Configurator.....	64
9	AUTOSAR Standard Compliance.....	65
9.1	Limitations / Restrictions	65
9.2	Hardware Limitations	65

9.3	Vector Extensions	67
10	Glossary and Abbreviations	68
10.1	Glossary	68
10.2	Abbreviations	68
11	Contact.....	69

Illustrations

Figure 3-1	AUTOSAR 3.x Architecture Overview	10
Figure 3-2	AUTOSAR architecture.....	11
Figure 3-3	Interfaces to adjacent modules of the CAN.....	11
Figure 5-1	Include Structure (AUTOSAR)	28
Figure 7-1	Select OS Type.....	35

Tables

Table 2-1	Supported Hardware Overview	8
Table 4-1	Supported features	15
Table 4-2	Hardware mailbox layout	17
Table 4-3	Errors reported to DET	20
Table 4-4	API from which the Errors are reported.....	21
Table 4-5	Errors reported to DEM.....	22
Table 4-6	Hardware Loop Check	24
Table 5-1	Static files	27
Table 5-2	Generated files	27
Table 5-3	Critical Section Codes	30
Table 5-4	Compiler abstraction and memory mapping.....	31
Table 7-1	MCAN CanIsr_<x>.....	36
Table 7-2	Can_InitMemory	37
Table 7-3	Can_InitController.....	38
Table 7-4	Can_InitController.....	39
Table 7-5	Can_ChangeBaudrate	39
Table 7-6	Can_CheckBaudrate	40
Table 7-7	Can_SetBaudrate	41
Table 7-8	Can_InitStruct.....	41
Table 7-9	Can_GetVersionInfo	42
Table 7-10	Can_GetStatus	43
Table 7-11	Can_SetControllerMode	44
Table 7-12	Can_ResetBusOffStart	44
Table 7-13	Can_ResetBusOffEnd	45
Table 7-14	Can_Write.....	46
Table 7-15	Can_CancelTx.....	46
Table 7-16	Can_CheckWakeup.....	47
Table 7-17	Can_DisableControllerInterrupts.....	47
Table 7-18	Can_EnableControllerInterrupts.....	48
Table 7-19	Can_MainFunction_Write	48
Table 7-20	Can_MainFunction_Read	49
Table 7-21	Can_MainFunction_BusOff.....	50
Table 7-22	Can_MainFunction_Wakeup.....	50
Table 7-23	Can_MainFunction_Mode.....	51

Table 7-24	Appl_GenericPreCopy	51
Table 7-25	Appl_GenericConfirmation	52
Table 7-26	Appl_GenericConfirmation	53
Table 7-27	Appl_GenericPreTransmit	53
Table 7-28	ApplCanTimerStart	54
Table 7-29	ApplCanTimerLoop	55
Table 7-30	ApplCanTimerEnd	55
Table 7-31	ApplCanInterruptDisable	56
Table 7-32	ApplCanInterruptRestore	57
Table 7-33	Appl_CanOverrun	57
Table 7-34	Appl_CanFullCanOverrun	58
Table 7-35	Appl_CanCorruptMailbox	59
Table 7-36	Appl_CanRamCheckFailed	59
Table 7-37	ApplCanInitPostProcessing	60
Table 7-38	Services used by the CAN	61
Table 10-1	Glossary	68
Table 10-2	Abbreviations	68

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
R7F701325A R7F701327 R7F701328 R7F701329	GHS Compiler Release v2015.1.7	Document Number: RH850/P1x-C Group Rev. 0.60, 09/2014	Rev. 0.60, Sep. 2014
R7F701370A R7F701370B R7F701371 R7F701372 R7F701372A R7F701373 R7F701373A R7F701374 R7F701374A		RH850/P1x-C Group Rev.0.10 , Nov. 2014	Nov, 2014 Rev.0.10
		RH850/P1x-C Group User’s Manual: Hardware Renesas microcontroller RH850 Family	Jan, 2016 Rev.1.00

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: (Supported AUTOSAR Standard Conform Features)	Pre-Compile, Link-Time, Post-Build Loadable, Post-Build Selectable (MICROSAR Identity Manager)	
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 CAN module 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.

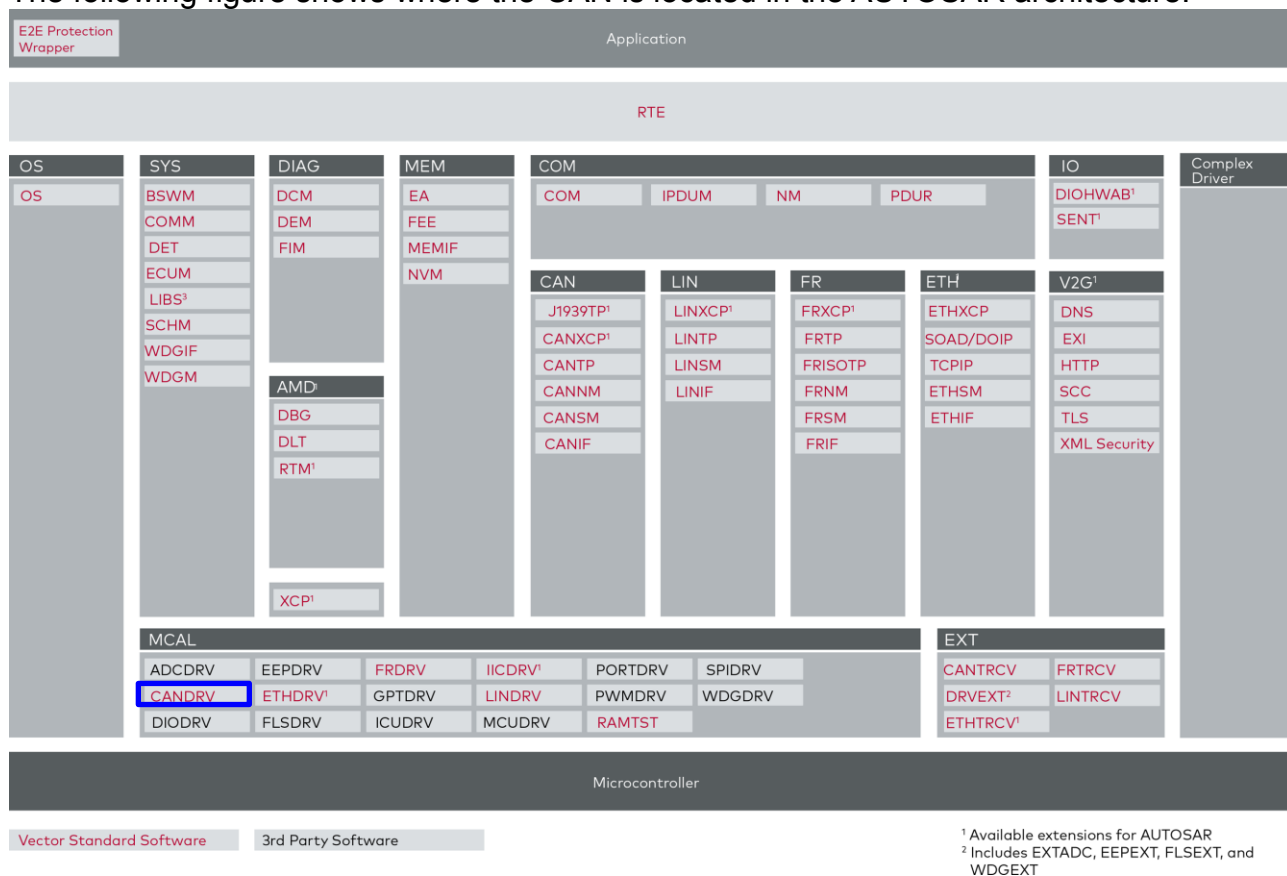


Figure 3-1 AUTOSAR 3.x Architecture Overview

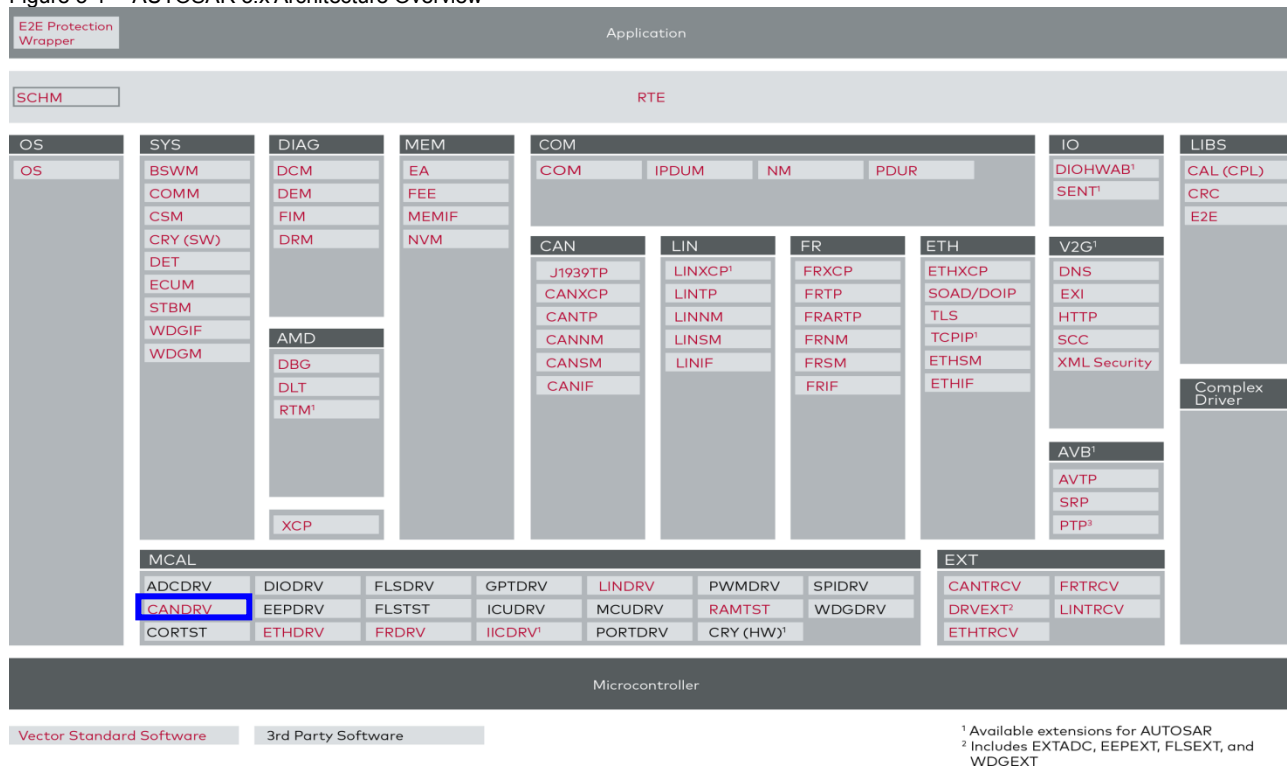


Figure 3-2 AUTOSAR architecture

The next figure shows the interfaces to adjacent modules of the CAN. These interfaces are described in chapter 7.

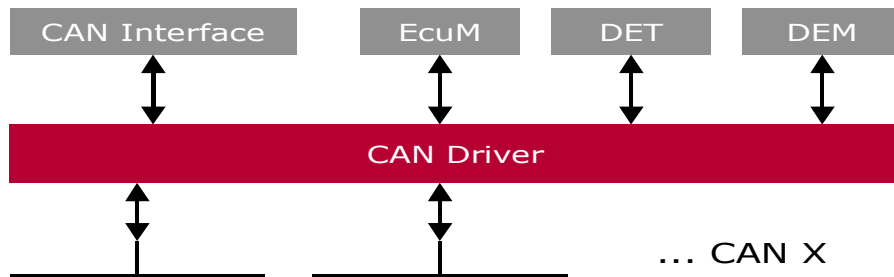


Figure 3-3 Interfaces to adjacent modules of the CAN

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 9.

Feature Naming	Short Description	CFG5
Initialization		
Driver	General driver initialization function Can_Init()	■
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.	32
Rx FullCAN	Separate mailbox for special Rx message used.	■
Maximum amount	Available amount of mailboxes.	64
Rx BasicCAN	Standard mailbox to receive CAN frame (depending on hardware, FIFO or shadow buffer supported).	■
Maximum amount	Available amount of BasicCAN objects. By default there is one FIFO(0) supported with a max. amount of 64 entries. In case of "Multiple BasicCAN" (see below) support an additional second FIFO(1) with 64 entries is supported.	2*64
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).	4
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 Rx Basic CAN	Support Multiple BasicCAN objects.	■ *

Multiple Tx Basic CAN	<p>This gives the possibility to use additionally Fifo-1 with 64 additional elements. By optimizing the acceptance filtering overruns can be avoided .</p> <p>Support Multiple Tx BasicCAN objects. Used to send different Tx groups over separate mailboxes with different buffering behavior (see Can Interface).</p>	■ *
Rx Queue	Support Rx Queue. This offers the possibility to buffer received data in interrupt context but handle it later asynchronous in the 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.	■
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 8-bit or 16-bit Hardware Handles depending on the 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	<input type="checkbox"/>

objects		
Overrun Notification	<p>Support DET or Application notification caused by overrun (overwrite) of an Rx message.</p> <p>Please note that 'Overrun' is supported for BasicCAN objects but is not available for FullCAN objects.</p> <p>While not processed a Message ID Filter Element referencing a specific FullCAN object will not match, causing the acceptance filtering to continue. Subsequent Message ID Filter Elements may cause the received message to be stored into</p> <ul style="list-style-type: none"> - another FullCAN object, or - a BasicCAN object, or - the message may be rejected, depending on the filter configuration. 	■
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 or cannot be configured for AutoSar version 4

**** Only available for MicroSar 4

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 type depends on the used mailboxes).

Hardware object number	Hardware object type	Amount of hardware objects	Description
0... N	Tx FullCAN	0 ... 31 max. (0 ... 29 in case of multiplexed transmission)	These objects are used to transmit specific message IDs. The user must define statically in the generation tool which CAN message IDs are located in Tx FullCAN objects. The generation tool assigns the message IDs to the FullCAN hardware objects.
(N+1) ... M	Tx BasicCAN	1 or 3 (3 in case of multiplexed transmission)	All other CAN message IDs are transmitted via the Tx Basic object. If the transmit message object is busy, the transmit requests are stored in the CAN Interface queue (if activated).
(M+1) ... O	Unused	0 ... 95	These objects are not used. It depends on the configuration of receive and transmit objects how many unused objects are available.
O...P	Rx FullCAN	0 ... 64	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 messages to the FullCAN objects.
96	Rx BasicCAN	FIFO-0 with max. 64 entries	All CAN message IDs, depending on the acceptance filter match, are received via the Rx BasicCAN message object through Rx FIFO 0. Each Rx Basic message object consists of 64 message buffers. 128 acceptance filters are available for standard IDs and 64 acceptance filters are available for extended IDs. In case of mixed ID mode 128+64 = 192 filters are available. Please note that this maximum amount of filters is also used for FIFO-1 if available.
97	Rx BasicCAN	FIFO-1 with max. 64 entries	All CAN message IDs, depending on the acceptance filter match, are received via the Rx BasicCAN message objects through Rx FIFO 1. Each Rx Basic message object consists of 64 message buffers. 128 acceptance filters are available for standard IDs and 64 acceptance filters are available for extended IDs. In case of mixed ID mode 128+64 = 192 filters are available. Please note that this maximum amount of filters is also used for FIFO-0.

Table 4-2 Hardware mailbox layout

The “CanObjectId” (ECUc parameter) numbering is done in following order: Tx FullCAN, Tx BasicCAN, Unused, Rx BasicCAN (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
Tx FullCAN	Object ID Low to High
Tx BasicCAN	Object ID Low to High
Rx FullCAN	Object ID Low to High
Rx BasicCAN	FIFO

In Case of Interrupt Rx FullCANs will be processed before Rx BasicCANs.

In Case of Polling Rx FullCANs will be processed before Rx 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

For each CAN channel a maximum amount of 128 filters for standard and 64 filters for extended ID configurations is available. Thus 192 filters are available for mixed ID configurations.

For acceptance filtering each list of filters is executed from element #0 until the first matching element. Acceptance filtering stops at the first matching element. Each filter element decides if the received message is stored within FIFO-0 (or FIFO-1 if available).

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

The CAN driver initializes the CAN controller not to receive remote frames. Therefore no additional action is required during runtime by the CAN driver for remote frame filtering. Remote frames will not have any influence on communication because they are not received by the CAN hardware.

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.

The following modes changes are supported:

`CAN_T_START`

`CAN_T_STOP`

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.

The Bit Stream Processor synchronizes itself to the data transfer on the CAN bus by waiting for the occurrence of a sequence of 11 consecutive recessive bits (= `Bus_Idle`) before it can take part in bus activities and start the message transfer.

4.4.2 Stop Mode

If stop mode is requested, either by software or by going `BusOff`, then the CAN module is switched into INIT mode. In this mode message transfer from and to the CAN bus is stopped, the status of the CAN bus transmit output is recessive (HIGH). Going to stop mode does not change any configuration register.

4.4.3 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.

4.7 Main Functions

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

If the Rx Queue feature is activated then 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() can be called by upper layers 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 gets an illegal handle as parameter
CAN_E_PARAM_DLC	API gets an illegal DLC as parameter
CAN_E_PARAM_CONTROLLER	API gets 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_DATALOSS (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. Avoid this by increasing the 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 are 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

As AUTOSAR extension the overrun detection may be activated by configuration tool. The notification can be configured to issue a DET call (MICROSAR 4.x) or an Application call (*Appl_CanOverrun()*).

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 “4.8.2.1”).

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
kCanLoopInit	<p>This channel dependent loop is called in Can_InitController and is processed as long as the CAN cell does not enter resp. leave the configuration mode.</p> <p>While entering the configuration mode, message transfer from and to the CAN bus is stopped, the status of the CAN bus transmit output is recessive.</p> <p>There is a delay from writing to a command register until the update of the related status register bits due to clock domain crossing (Host and CAN clock). Therefore the programmer has to assure that the previous value written to INIT has been accepted.</p> <p>Due to the high precision clocking requirements of the CAN Core, a separate clock without any modulation has to be provided as CAN clock. The CAN Core should be programmed to have at least 8 clocks per bit time (e.g.: at least 8 MHz CAN clock at 1 Mbaud CAN speed). In order to achieve a stable function of the M_CAN, the Host clock must always be faster than or equal to the CAN clock.</p> <p>If the loop cancels, try to reinitialize the controller again or reset the hardware.</p> <p>After leaving the configuration mode the Bit Stream Processor synchronizes itself to the data transfer on the CAN bus by waiting for the occurrence of a sequence of 11 consecutive recessive bits (= Bus_Idle) before it can take part in bus activities and start the message transfer.</p>
kCanLoopStart	<p>MICROSAR3:</p> <ul style="list-style-type: none"> - Used while transition in mode 'START'. - Call context: Can_SetControllerMode() - There is a delay from writing to a command register until the update of the related status register bits due to clock domain crossing (Host and CAN clock). Therefore the programmer has to assure that the previous value written to INIT has been accepted. - If the loop cancels try to recall Can_SetControllerMode().

Loop Name / source	Short Description
	<p>MICROSAR4:</p> <p>Used for short time mode transition blocking (short synchronous timeout). Same value for kCanLoopStart, kCanLoopStop, kCanLoopSleep and kCanLoopWakeup.</p> <p>No Issue when timeout occurs.</p>
kCanLoopStop	<p>MICROSAR3:</p> <ul style="list-style-type: none"> - Used while transition in mode 'STOP'. - Call context: Can_SetControllerMode() - There is a delay from writing to a command register until the update of the related status register bits due to clock domain crossing (Host and CAN clock). Therefore the programmer has to assure that the previous value written to INIT has been accepted. - If the loop cancels try to recall Can_SetControllerMode(). <p>MICROSAR4:</p> <p>Used for short time mode transition blocking (short synchronous timeout). Same value for kCanLoopStart, kCanLoopStop, kCanLoopSleep and kCanLoopWakeup.</p> <p>No Issue when timeout occurs.</p>
kCanLoopSleep	<p>MICROSAR3:</p> <ul style="list-style-type: none"> - Used while transition in mode 'SLEEP'. - Call context: Can_SetControllerMode() - When all pending transmission requests have completed, the M_CAN waits until bus idle state is detected. - If the loop cancels try to recall Can_SetControllerMode. <p>MICROSAR4:</p> <p>Used for short time mode transition blocking (short synchronous timeout). Same value for kCanLoopStart, kCanLoopStop, kCanLoopSleep and kCanLoopWakeup.</p> <p>No Issue when timeout occurs.</p>
kCanLoopWakeup	<p>MICROSAR3:</p> <ul style="list-style-type: none"> - Used while transition in mode 'WAKEUP'. - Call context: Can_SetControllerMode() - Once the M_CAN is initialized it synchronizes itself to the CAN bus and is ready for communication. - If the loop cancels try to recall Can_SetControllerMode. <p>MICROSAR4:</p> <p>Used for short time mode transition blocking (short synchronous timeout). Same value for kCanLoopStart, kCanLoopStop, kCanLoopSleep and kCanLoopWakeup.</p> <p>No Issue when timeout occurs.</p>

kCanLoopClock Stop	When Clock Stop is requested then all pending transfer requests are completed first. When the CAN bus reached idle then Clock Stop will be acknowledged.
kCanLoopRxFifo	This channel dependent loop is called in CanInterruptRxFifo and is processed until the Rx FIFO becomes empty. The loop is delayed if the controller receives a burst of messages. The maximum expected duration is the time needed until all messages in the reception FIFO are confirmed. If the loop cancels, reinitialize the Controller.

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().

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.



Caution

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.

4.9.1 Error Interrupt

The MCAN error interrupt source is used only partially by the CAN driver. Only BusOff events are handled and reported to the upper layers by the CAN driver.

Not reported errors are:

Stuff Error	More than 5 equal bits in a sequence occurred
Format Error	A fixed format part of a received frame has the wrong format
Acknowledge Error	A transmitted message was not acknowledged by another node
Bit Error	Device wanted to send a recessive/dominant level, but the monitored level was dominant/recessive
CRC Error	Received CRC did not match the calculated CRC
Watchdog Interrupt	Message RAM Watchdog event due to missing READY
Warning Status	Error_Warning status changed
Error Passive	Error_Passive status changed
Error Logging Overflow	Overflow of CAN Error Logging Counter occurred
Bit Error Uncorrected	Message RAM bit error detected, uncorrected.
Bit Error Corrected	Message RAM bit error detected and corrected.
Timeout Occurred	Timeout reached
Timestamp	Wraparound Timestamp counter wrapped around
Rx FIFO 0 Full	Rx FIFO 0 Full
Rx FIFO 0 Watermark	Reached fill level watermark
Rx FIFO 1 Full	Rx FIFO 1 Full
Rx FIFO 1 Watermark	Reached fill level watermark

**Please note**

The BusOff recovery sequence cannot be shortened (e.g. by initializing the CAN device). If the device goes BusOff, it will enter the INIT mode by its own, stopping all bus activities.

When leaving the INIT mode the device will wait for 129 occurrences of Bus Idle (129 x 11 consecutive recessive bits) before resuming normal operation.

**Please note**

The Timeout Counter is used for CAN driver internal purposes (supervision of possible transmit confirmations arriving delayed after a cancellation was requested). Thus the "Timeout Occurred" interrupt may occur occasionally.

4.9.2 Not supported

Neither the Tx Event FIFO nor the Tx Queue is used. All available 32 transmit message buffers per CAN channel are used as dedicated buffers and can be used either as BasicCAN or FullCAN objects (see 4.3.1).

The filtering of High Priority messages is not supported.

No Range Filters are supported.

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.(lib)	This is the library build out of Can.c, Can.h and Can_Local.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. (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

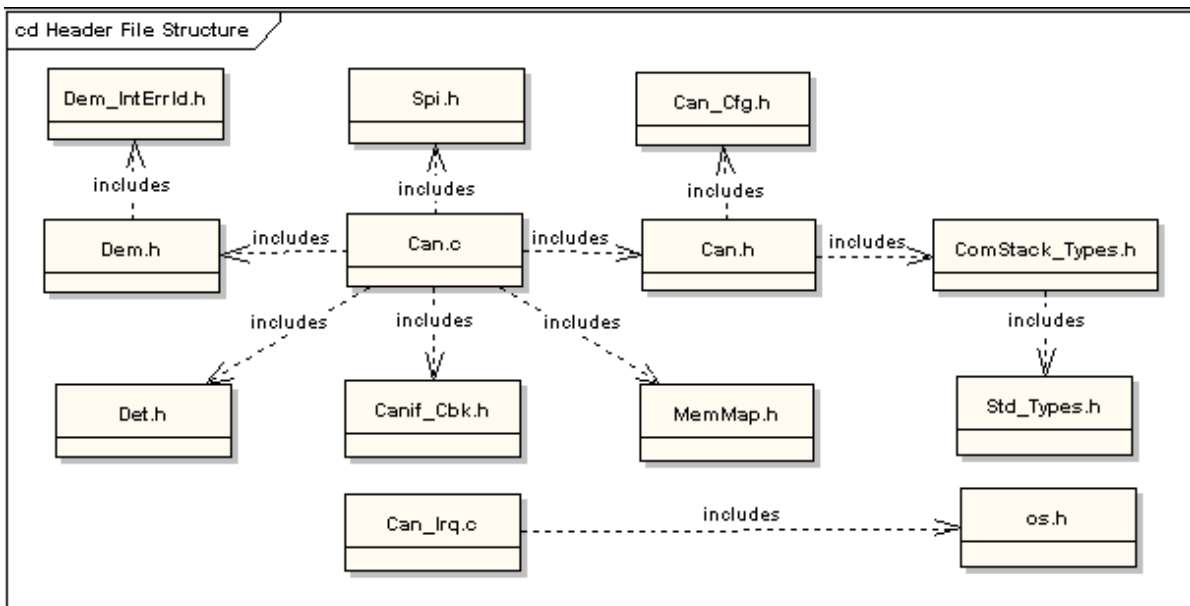


Figure 5-1 Include Structure (AUTOSAR)

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.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>CanNestedGlobalInterruptDisable/Restore() is used within Can_MainFunction_Write() to assure that transmit confirmations do not conflict with further transmit requests.</p> <ul style="list-style-type: none"> > Duration is short. > 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. > 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 do not call function Can_Write() 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 avoid nested state transition requests.</p> <ul style="list-style-type: none"> > Duration is medium > No API call of other BSW inside. > Use CAN interrupt locks here, in case the above mentioned APIs are only called within same tasklevel and CAN interrupt context (no nesting - like BusOff-handling in interrupt has to be blocked). 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.

Compiler Abstraction Definitions	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_CANCEL`: is used to access the CAN cell itself.

`CAN_ RX_TX_DATA`: access to CAN Data buffers.

`CAN_ APPL_*`: access to higher layers.

6 Hardware Specific Hints

6.1.1 Usage of interrupt functions

According to the current implementation of MCAN generator there is a fix assignment of interrupt functions to the CAN Controller. The postfix of the interrupt function name equates the controller number. The following table shows the corresponding assignment for the derivative RH850 P1X-C.

Critical Section Define	Description
MCAN_0, BaseAddress: 0xFFEF0000 CanIsr_1	CanIsr_1
MCAN_1, BaseAddress: 0xFFD31000 CanIsr_2	CanIsr_2
MCAN_2, BaseAddress: 0xFFEF1000 CanIsr_3	CanIsr_3


Table 5-5 Hardware Controller – Interrupt Functions


CanIsr_0 is used for MTT_CAN0 of the RH850 P1X-C.

6.1.2 MCAN Errata

The following Errata (please see [6] for further details) are considered by the CAN Driver. By default all erratas which are appropriate for the configured MCAN Revision are enabled. If a specific erratum shall be disabled or enabled beyond that it can be configured via a user configuration file.

Errata No.	Title	MCAN Rev. affected
6	Change of CAN operation mode during start of transmission. Only activated if "CAN_BOSCH_ERRATUM_006" is defined as STD_ON.	2.9.5, 2.9.6, 3.0.0, 3.0.1
7	Problem with frame transmission after recovery from Restricted Operation Mode. Only activated if "CAN_BOSCH_ERRATUM_007" is defined as STD_ON.	2.9.5, 2.9.6, 3.0.0, 3.0.1
8	Setting / resetting CCCR.INIT during frame reception. Only activated if "CAN_BOSCH_ERRATUM_008" is defined as STD_ON.	2.9.5, 2.9.6, 3.0.0, 3.0.1
10	Setting CCCR.CCE while a Tx scan is ongoing. Only activated if "CAN_BOSCH_ERRATUM_010" is defined as STD_ON.	2.9.5, 2.9.6, 3.0.0,

		3.0.1
11	<p>Needless activation of interrupt IR.MRAF.</p> <p>Only activated if "CAN_BOSCH_ERRATUM_011" is defined as STD_ON.</p>	2.9.5, 2.9.6, 3.0.0, 3.0.1, 3.1.0
12	<p>Return of receiver from Bus Integration state after Protocol Exception Event.</p> <p>Only activated if "CAN_BOSCH_ERRATUM_012" is defined as STD_ON.</p>	2.9.6, 3.0.0, 3.0.1, 3.1.0
13	<p>Message RAM / RAM Arbiter not responding in time.</p> <p>When the M_CAN wants to store a received frame and the Message RAM / RAM Arbiter does not respond in time, this message cannot be stored completely and it is discarded with the reception of the next message. Interrupt flag IR.MRAF is set. It may happen that the next received message is stored incomplete.</p> <p>In this case, the respective Rx Buffer or Rx FIFO element holds inconsistent data.</p> <div style="background-color: #f0f0f0; padding: 10px; margin-top: 10px;">  <p>When the M_CAN has been integrated correctly (the Host and the CAN clock must be fast enough to handle a worst case configuration containing the maximum of MCAN Message RAM elements), this behaviour can only occur in case of a problem with the Message RAM itself or the RAM Arbiter.</p> <p>The application must assure that the clocking of Host and CAN is appropriate. The CAN Driver does not care about these configuration aspects.</p> </div>	2.9.6, 3.0.0, 3.0.1, 3.1.0, 3.2.0

14	<p>Data loss (payload) in case storage of a received frame has not completed until end of EOF field is reached.</p> <p>The time needed for acceptance filtering and storage of a received message depends on the</p> <ul style="list-style-type: none"> - Host clock frequency, - the number of M_CANs connected to a single Message RAM, - the Message RAM arbitration scheme, and - the number of configured filter elements. <p>In case storage of a received message has not completed until end of the received frame then corrupted data can be contained in the Message RAM.</p> <p>Interrupt flag IR.MRAF is not set.</p> <div style="background-color: #e6f2ff; padding: 10px; margin-top: 10px;">  <p>If storage of messages cannot be completed the application is responsible for reducing the maximum number of configured filter elements for the M_CANs attached to the Message RAM until the calculated clock frequency is below the Host clock frequency used with the actual device.</p> </div>	2.9.6, 3.0.0, 3.0.1, 3.1.0, 3.2.0
1-5	These errata are in the responsibility of the application and are not considered by the CAN Driver.	2.0.0, 2.9.5, 2.9.6, 3.0.0, 3.0.1
9	<p>Frame transmission in DAR mode.</p> <p>Not considered by the CAN Driver, frame transmission in DAR mode is not supported.</p>	2.9.5, 2.9.6, 3.0.0, 3.0.1
15	<p>Edge filtering causes mis-synchronization when falling edge at Rx input pin coincides with end of integration phase.</p> <p>Not considered by the CAN Driver, Edge Filtering is not supported.</p>	3.1.0, 3.2.0, 3.2.1
16	<p>Configuration of NBTP.NTSEG2 = '0' not allowed.</p> <p>Not considered by the CAN Driver, the user is responsible to care about the according bit timing configuration.</p>	3.1.0, 3.2.0, 3.2.1

7 API Description

7.1 Interrupt Service Routines provided by CAN

Depend on the settings in Tools component Hw_Mpc5700Cpu, the interrupt routine is given by the driver or by Operating System. (Selection below, not MICROSAR403)

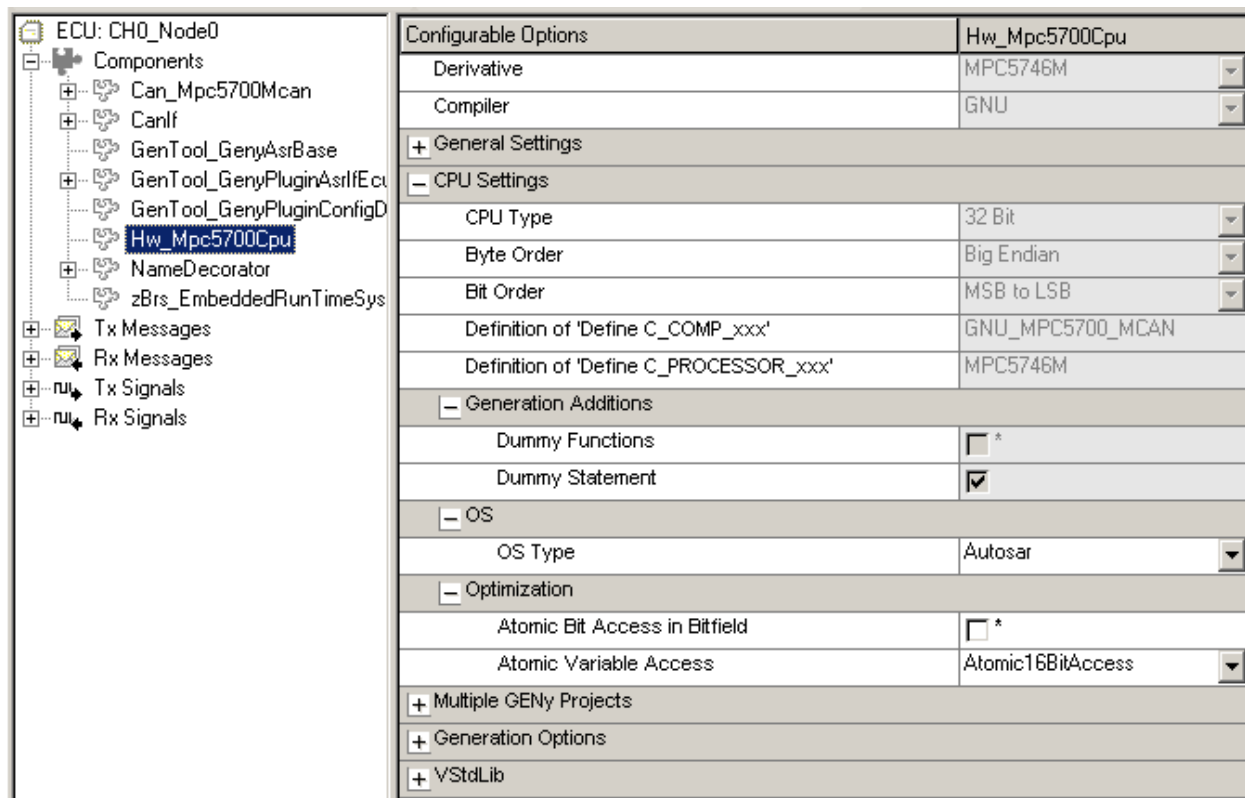


Figure 7-1 Select OS Type

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.

7.1.1 OSEK (OS)

This means to include osek.h.

Switch: V_OSTYPE_OSEK

7.1.2 AutoSar (OS)

Os.h header file is used.

Switch: V_OSTYPE_AUTOSAR

7.1.3 None (OS)

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

Switch: V_OSTYPE_NONE

7.1.4 Type of Interrupt Function

- 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).
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

7.1.5 CAN ISR API

Prototype	
void CanIsr_<x>(void);	
Parameter	
---	---
Return code	
---	---
Functional Description	
Handles interrupts of hardware channel <x> for Rx, Tx, BusOff events.	
Particularities and Limitations	
<ul style="list-style-type: none"> > Number of available functions depends on used MCU derivative. > The functions are not designated as interrupt functions. If it is necessary to save/restore all general purpose registers and to use a different "return from interrupt" instruction the application code has to implement the compiler specific pragma (e.g. for Wind River™ DIAB™: #pragma interrupt CanIsr_x). 	

Table 7-1 MCAN CanIsr_<x>

7.2 Services provided by CAN

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

7.2.1 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	
<ul style="list-style-type: none"> > 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 7-2 Can_InitMemory

7.2.2 Can_Init

Prototype	
void Can_Init(const Can_ConfigType *Config)	
Parameter	
Config	<p>Pointer to the structure including configuration data.</p> <p>In case of Multiple ECU configuration feature is used, for each Identity one “Config” structure exists and has to be chosen here</p>
Return code	
-	-
Functional Description	
This function initializes global CAN driver variables during ECU start-up.	
Particularities and Limitations	
<ul style="list-style-type: none"> > 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. 	

7.2.3 Can_InitController

Prototype	
void Can_InitController (uint8 Controller, Can_ControllerBaudrateConfigPtrType Config)	
Parameter	
Controller [in]	Number of controller
Config [in]	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	
<p>Called by CanInterface.</p> <p>Disabled Interrupts.</p>	
Call context	
<ul style="list-style-type: none"> > Must be called during the startup sequence before CAN communication takes place but after calling „Can_Init()“. > Must not be called while in „Sleep Mode“. > This function is Synchronous > This function is Non-Reentrant > Availability: MICROSAR401 only 	

Table 7-3 Can_InitController

7.2.4 Can_InitController

Prototype	
void Can_InitController (uint8 Controller, Can_ControllerConfigPtrType ControllerConfigPtr)	
Parameter	
Controller [in]	Number of controller
Config [in]	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</p>	

Mode".
Particularities and Limitations
Called by CanInterface. Disabled Interrupts
Call context
<ul style="list-style-type: none"> > Must be called during the startup sequence before CAN communication takes place but after calling „Can_Init()“. > Must not be called while in „Sleep Mode“. > This function is Synchronous > This function is Non-Reentrant > Availability: MICROSAR3 only

Table 7-4 Can_InitController

7.2.5 Can_ChangeBaudrate

Prototype	
Std_ReturnType Can_ChangeBaudrate (uint8 Controller, const uint16 Baudrate)	
Parameter	
Controller [in]	Number of controller to be changed
Baudrate [in]	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. The CAN controller must be in "Stop Mode".	
Call context	
<ul style="list-style-type: none"> > Must be called during the startup sequence before CAN communication takes place but after calling „Can_Init()“. > This function is Synchronous > This function is Non-Reentrant > Availability: MICROSAR403 only & if „CanChangeBaudrateApi“ is activated or „CanSetBaudrateApi“ is de-activated. 	

Table 7-5 Can_ChangeBaudrate

7.2.6 Can_CheckBaudrate

Prototype	
Std_ReturnType Can_CheckBaudrate (uint8 Controller, const uint16 Baudrate)	
Parameter	
Controller [in]	Number of controller to be checked
Baudrate [in]	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	
<p>Called by Application.</p> <p>The CAN controller must be initialized.</p>	
Call context	
<ul style="list-style-type: none"> > Must not be called nested. > Only available if „CanChangeBaudrateApi“ is activated. > This function is Synchronous > This function is Non-Reentrant > Availability: MICROSAR403 only & „CanChangeBaudrateApi“ is activated („CAN_CHANGE_BAUDRATE_SUPPORT == STD_ON“) 	

Table 7-6 Can_CheckBaudrate

7.2.7 Can_SetBaudrate

Prototype	
Std_ReturnType Can_SetBaudrate (uint8 Controller, uint16 BaudRateConfigID)	
Parameter	
Controller [in]	Number of controller to be set
BaudRateConfigID [in]	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 7-7 Can_SetBaudrate

7.2.8 Can_InitStruct


Prototype	
void Can_InitStruct (uint8 Controller, uint8 Index)	
Parameter	
Controller [in]	Number of the controller to be changed
Index [in]	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	
<p>Called by Application.</p> <p>“Can_Init” was called.</p>	
	<p>Caution</p> <p>None AUTOSAR API</p>
Call context	
<ul style="list-style-type: none"> > Call this function between calling „Can_Init()“ and „Can_InitController()“. > This function is Synchronous > This function is Non-Reentrant > Availability: MICROSAR3 only 	

Table 7-8 Can_InitStruct

7.2.9 Can_GetVersionInfo

Prototype
void Can_GetVersionInfo (Can_VersionInfoPtrType VersionInfo)

Parameter	
VersionInfo [out]	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	
<ul style="list-style-type: none"> > Only available if „CanVersionInfoApi“ is activated. > This function is Synchronous > This function is Reentrant > Availability: „CanVersionInfoApi“ is activated („CAN_VERSION_INFO_API == STD_ON“) 	

Table 7-9 Can_GetVersionInfo

7.2.10 Can_GetStatus

Prototype	
uint8 Can_GetStatus (uint8 Controller)	
Parameter	
Controller [in]	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.	


<p>Only one of the status bits CAN_STATUS_SLEEP/STOP/BUSOFF/PASSIVE/WARNING is set.</p> <p>The CAN_STATUS_INIT bit is always set if a controller is initialized.</p> <p>CAN_STATUS_SLEEP has the highest and CAN_STATUS_WARNING the lowest priority.</p> <p>CAN_STATUS_INCONSISTENT will be set if one Common CAN channel. Is not "Stop" or "Sleep".</p> <p>CAN_DEACTIVATE_CONTROLLER is set in case the "CanRamCheck" detected an Issue.</p> <p>"status" can be analyzed using the provided API macros:</p> <p>CAN_HW_IS_OK(status): return "true" in case no warning, passive or bus off occurred.</p> <p>CAN_HW_IS_WARNING(status): return "true" in case of waning status.</p> <p>CAN_HW_IS_PASSIVE(status): return "true" in case of passive status.</p> <p>CAN_HW_IS_BUSOFF(status): return "true" in case of bus off status (may be already false in Notification).</p> <p>CAN_HW_IS_WAKEUP(status): return "true" in case of not in sleep mode.</p> <p>CAN_HW_IS_SLEEP(status): return "true" in case of sleep mode.</p> <p>CAN_HW_IS_STOP(status): return "true" in case of stop mode.</p> <p>CAN_HW_IS_START(status): return "true" in case of not in stop mode.</p> <p>CAN_HW_IS_INCONSISTENT(status): return "true" in case of an inconsistency between two common CAN channels.</p>	
Particularities and Limitations	
Called by network management or Application.	
	Caution None AUTOSAR API
Call context	
<ul style="list-style-type: none">> This function is Synchronous> This function is Non-Reentrant> Availability: „CanGetStatus“ is activated („CAN_GET_STATUS == STD_ON“)	

Table 7-10 Can_GetStatus

7.2.11 Can_SetControllerMode

Prototype	
Can_ReturnType Can_SetControllerMode (uint8 Controller, Can_StateTransitionType Transition)	
Parameter	
Controller [in]	Number of the controller to be set
Transition [in]	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. Interrupts locked by CanInterface
Call context
<ul style="list-style-type: none"> > Must not be called within CAN driver context like RX, TX or Bus Off callouts. > This function is Non-Reentrant > Availability: Always

Table 7-11 Can_SetControllerMode

7.2.12 Can_ResetBusOffStart


Prototype	
void Can_ResetBusOffStart (uint8 Controller)	
Parameter	
Controller [in]	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	
<ul style="list-style-type: none"> > Called while BusOff event handling (Polling or Interrupt context). > This function is Synchronous > This function is Non-Reentrant > Availability: Always 	

Table 7-12 Can_ResetBusOffStart

7.2.13 Can_ResetBusOffEnd

Prototype
void Can_ResetBusOffEnd (uint8 Controller)


Parameter	
Controller [in]	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	
<ul style="list-style-type: none"> > Called inside „Can_SetControllerMode()“ while Start transition. > This function is Synchronous > This function is Non-Reentrant > Availability: Always 	

Table 7-13 Can_ResetBusOffEnd

7.2.14 Can_Write

Prototype	
Can_ReturnType Can_Write (Can_HwHandleType Hth, Can_PduInfoPtrType PduInfo)	
Parameter	
Hth [in]	Handle of the mailbox intended to send the message
PduInfo [in]	Information about the outgoing message (ID, dataLength, 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 controller is busy.
Functional Description	
Send a CAN message over CAN.	
Particularities and Limitations	
Called by CanInterface. CAN Interrupt locked.	
Call context	
<ul style="list-style-type: none"> > Called by the CanInterface with at least disabled CAN interrupts. > (Due to data security reasons the CanInterface should accomplish this and thus it is not needed further more in the CAN Driver.) 	

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

Table 7-14 Can_Write

7.2.15 Can_CancelTx


Prototype	
void Can_CancelTx (Can_HwHandleType Hth, PduIdType PduId)	
Parameter	
Hth [in]	Handle of the mailbox intended to be cancelled.
PduId [in]	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	
<ul style="list-style-type: none"> > Called by CanTp or Application. > This function is Synchronous > This function is Non-Reentrant > Availability: Always 	

Table 7-15 Can_CancelTx

7.2.16 Can_CheckWakeup

Prototype	
Std_ReturnType Can_CheckWakeup (uint8 Controller)	
Parameter	
Controller [in]	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
<ul style="list-style-type: none"> > Called while Wakeup validation phase. > 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 7-16 Can_CheckWakeup

7.2.17 Can_DisableControllerInterrupts

Prototype	
void Can_DisableControllerInterrupts (uint8 Controller)	
Parameter	
Controller [in]	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.	
Must not be called while CAN controller is in sleep mode.	
Call context	
<ul style="list-style-type: none">> Called within Critical Area handling or out of Application code.> This function is Synchronous> This function is Non-Reentrant> Availability: Always	

Table 7-17 Can_DisableControllerInterrupts

7.2.18 Can_EnableControllerInterrupts

Prototype
<code>void Can_EnableControllerInterrupts (uint8 Controller)</code>

Parameter	
Controller [in]	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. Must not be called while CAN controller is in sleep mode.	
Call context	
<ul style="list-style-type: none"> > Called within Critical Area handling or out of Application code. > This function is Synchronous > This function is Non-Reentrant > Availability: Always 	

Table 7-18 Can_EnableControllerInterrupts

7.2.19 Can_MainFunction_Write

Prototype	
void Can_MainFunction_Write (void)	
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. Must not interrupt the call of "Can_Write()".	
Call context	
<ul style="list-style-type: none"> > Called within cyclic TX task. > This function is Synchronous > This function is Non-Reentrant > Availability: Always 	

Table 7-19 Can_MainFunction_Write

7.2.20 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	
<ul style="list-style-type: none"> > Called within cyclic RX task. > This function is Synchronous > This function is Non-Reentrant > Availability: Always 	

Table 7-20 Can_MainFunction_Read

7.2.21 Can_MainFunction_BusOff

Prototype	
<code>void Can_MainFunction_BusOff (void)</code>	
Parameter	
-	
Return code	
void	-
Functional Description	
<p>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).</p>	
Particularities and Limitations	
Called by SchM.	
Call context	
<ul style="list-style-type: none"> > Called within cyclic BusOff task. > This function is Synchronous > This function is Non-Reentrant 	

> Availability: Always

Table 7-21 Can_MainFunction_BusOff

7.2.22 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	
<ul style="list-style-type: none"> > Called within cyclic Wakeup task. > This function is Synchronous > This function is Non-Reentrant > Availability: Always 	

Table 7-22 Can_MainFunction_Wakeup

7.2.23 Can_MainFunction_Mode

Prototype	
void Can_MainFunction_Mode (void)	
Parameter	
-	
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.	
Call context	
<ul style="list-style-type: none"> > Called within cyclic mode change task. 	

- > This function is Synchronous
- > This function is Non-Reentrant
- > Availability: MICROSAR4x only

Table 7-23 Can_MainFunction_Mode

7.2.24 Appl_GenericPrecopy


Prototype	
Can_ReturnType Appl_GenericPrecopy (uint8 Controller, Can_IdType ID, uint8 DataLength, Can_DataPtrType DataPtr)	
Parameter	
Controller [in]	Controller which received the message
ID [in]	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. FD-bit will not be set at all.
DataLength [in]	Data length of the received message.
pData [in]	Pointer to the data of the received message.
Return code	
Can_ReturnType	<ul style="list-style-type: none"> > CAN_OK if the indication of the message should be called afterwards (notification to higher layer), > CAN_NOT_OK in case of stopping furthermore reception.
Functional Description	
Application callback function which informs about all incoming RX messages including the contained data.	
Particularities and Limitations	
Called by CAN driver.	
"pData" is read only and must not be accessed for further write operations.	
	Caution None AUTOSAR API
Call context	
<ul style="list-style-type: none"> > Called within CAN message reception context (Polling or Interrupt). > This function is Synchronous > This function is Non-Reentrant > Availability: „CanGenericPrecopy“ is activated („CAN_GENERIC_PRECOPY == STD_ON“). 	

Table 7-24 Appl_GenericPrecopy

7.2.25 Appl_GenericConfirmation


Prototype	
Can_ReturnType Appl_GenericConfirmation (PduIdType PduId)	
Parameter	
PduId [in]	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	
<p>Called by CAN driver.</p> <p>“PduId” is read only and must not be accessed for further write operations.</p>	
	<p>Caution</p> <p>None AUTOSAR API</p>
Call context	
<ul style="list-style-type: none"> > Called within CAN message transmission finished context (Polling or Interrupt). > This function is Synchronous > This function is Non-Reentrant > Availability: „CanGenericConfirmation“ is activated („CAN_GENERIC_CONFIRMATION == STD_ON“) & „CanIfTransmitBuffer“ activated (in CanInterface). 	

Table 7-25 Appl_GenericConfirmation

7.2.26 Appl_GenericConfirmation

Prototype	
Can_ReturnType Appl_GenericConfirmation (uint8 Controller, Can_PduInfoPtrType DataPtr)	
Parameter	
Controller [in]	Number of the causing controller.
DataPtr [in]	
Return code	
Can_ReturnType	<p>CAN_OK Higher layer (CanInterface) confirmation will be called.</p> <p>CAN_NOT_OK No further higher layer (CanInterface) confirmation will be called.</p>
Functional Description	
Application callback function which informs about TX messages being sent to the CAN bus.	


Particularities and Limitations	
<p>Called by CAN driver.</p> <p>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.</p>	
	<p>Caution</p> <p>None AUTOSAR API</p>
Call context	
<ul style="list-style-type: none"> > Called within CAN message transmission finished context (Polling or Interrupt). > This function is Synchronous > This function is Non-Reentrant > Availability: If "CanGenericConfirmation" ("CAN_GENERIC_CONFIRMATION == STD_ON") and "CanIfTransmitBuffer" (in CanInterface) is activated. 	

Table 7-26 Appl_GenericConfirmation

7.2.27 Appl_GenericPreTransmit


Prototype	
<pre>void Appl_GenericPreTransmit (uint8 Controller, Can_PduInfoPtrType_var DataPtr)</pre>	
Parameter	
Controller [in]	Number of the controller on which the hardware observation takes place.
DataPtr [in]	Pointer to a Can_PduType structure including ID, DataLength, 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.	
	<p>Caution</p> <p>None AUTOSAR API</p>
Call context	
<ul style="list-style-type: none"> > Called within „Can_Write()“. > This function is Synchronous > This function is Non-Reentrant > Availability: „CanGenericPretransmit“ is activated („CAN_GENERIC_PRETRANSMIT == STD_ON“). 	

Table 7-27 Appl_GenericPreTransmit

7.2.28 ApplCanTimerStart


Prototype	
<code>void ApplCanTimerStart (CanChannelHandle Controller, uint8 source)</code>	
Parameter	
Controller [in]	Number of the controller on which the hardware observation takes place. (only if not using "Optimize for one controller")
source [in]	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	
<ul style="list-style-type: none"> > 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 7-28 ApplCanTimerStart

7.2.29 ApplCanTimerLoop

Prototype	
<code>Can_ReturnType ApplCanTimerLoop (CanChannelHandle Controller, uint8 source)</code>	
Parameter	
Controller [in]	Number of the controller on which the hardware observation takes place. (only if not using "Optimize for one controller")
source [in]	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	
<ul style="list-style-type: none"> > 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 7-29 ApplCanTimerLoop

7.2.30 ApplCanTimerEnd


Prototype	
void ApplCanTimerEnd (CanChannelHandle Controller, uint8 source)	
Parameter	
Controller [in]	Number of the controller on which the hardware observation takes place. (only if not using “Optimize for one controller”)
source [in]	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	
<ul style="list-style-type: none"> > 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 7-30 ApplCanTimerEnd

7.2.31 ApplCanInterruptDisable


Prototype	
void ApplCanInterruptDisable (uint8 Controller)	
Parameter	
Controller [in]	Number of the controller for the CAN interrupt lock.
Return code	
void	-
Functional Description	
<p>Service function to support the disabling 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.	
	<p>Caution</p> <p>None AUTOSAR API</p>
Call context	
<ul style="list-style-type: none"> > Called by the CAN Driver within „Can_DisableControllerInterrupts()“. > 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 7-31 ApplCanInterruptDisable

7.2.32 ApplCanInterruptRestore

Prototype	
void ApplCanInterruptRestore (uint8 Controller)	
Parameter	
Controller [in]	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	
<ul style="list-style-type: none"> > Called by the CAN Driver within „Can_EnableControllerInterrupts()“. > 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 7-32 ApplCanInterruptRestore

7.2.33 Appl_CanOverrun


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

Table 7-33 Appl_CanOverrun

7.2.34 Appl_CanFullCanOverrun

Prototype
void Appl_CanFullCanOverrun (uint8 Controller)



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

Table 7-34 Appl_CanFullCanOverrun

7.2.35 Appl_CanCorruptMailbox

Prototype	
void Appl_CanCorruptMailbox (uint8 Controller, Can_HwHandleType hwObjHandle)	
Parameter	
Controller [in]	Number of the controller for which the check failed.
hwObjHandle [in]	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	
<ul style="list-style-type: none"> > Call within controller initialization. > This function is Synchronous > This function is Non-Reentrant 	

> Availability: „CanRamCheck“ set to „MailboxNotifiation“ („CAN_RAM_CHECK == CAN_NOTIFY_MAILBOX“).

Table 7-35 Appl_CanCorruptMailbox

7.2.36 Appl_CanRamCheckFailed


Prototype	
uint8 Appl_CanRamCheckFailed (uint8 Controller)	
Parameter	
Controller [in]	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.	
	<p>Caution</p> <p>None AUTOSAR API</p>
Call context	
<ul style="list-style-type: none"> > Call within controller initialization. > This function is Synchronous > This function is Non-Reentrant > Availability: „CanRamCheck“ set to „Active“ or „MailboxNotifiation“ („CAN_RAM_CHECK != CAN_NONE“). 	

Table 7-36 Appl_CanRamCheckFailed

7.2.37 ApplCanInitPostProcessing

Prototype	
void ApplCanInitPostProcessing (CAN_HW_CHANNEL_CANTYPE_ONLY)	
Parameter	
Controller [in]	Number of the controller for which the check failed
Return code	
void	none


Functional Description	
<p>Service function to overwrite the previously set initialization values for the bit timing, taken from the generated data, with customer specific values.</p> <p>For your convenience the following access function is supported:</p> <ul style="list-style-type: none"> - CanBtpReg(controller): - the BTP register of the specified CAN channel can be set according to the register definition as specified in the Hardware Manufacturer Document ((see ch. 2). <p>Example: CanBtpReg(Controller) = 0x00070F70u; or CanBtpReg(0) = 0x00070F70u; (when using 'Optimize for one controller').</p>	
Particularities and Limitations	
<p>Called by CAN driver.</p> <p>None</p>	
	<p>Caution</p> <p>None AUTOSAR API</p> <p>It is the responsibility of the application to assure that the register values are consistent with the release of the underlying derivative.</p>
Call context	
<ul style="list-style-type: none"> > Called within controller initialization. > This function is Synchronous > This function is Non-Reentrant > Availability: Only available if ,C_ENABLE_INIT_POST_PROCESS' is defined via a user-config file. 	

Table 7-37 ApplCanInitPostProcessing

7.3 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])

Component	API
Application (optional non AUTOSAR)	Appl_GenericPrecopy Appl_GenericConfirmation Appl_GenericPreTransmit ApplCanTimerStart/Loop/End Appl_CanRamCheckFailed, Appl_CanCorruptMailbox ApplCanInterruptDisable/Restore Appl_CanOverrun, For detailed description see Chapter 7.2
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 7-38 Services used by the CAN

8 Configuration

For CAN driver the attributes can be configured with configuration Tool “CFG5”

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 CFG5 configuration (see Chapter “DaVinci Configurator”).

8.1 Pre-Compile Parameters

Some settings have to be available before compilation:

- > MCAN Core Release
#define C_ENABLE_MPC5700_MCAN_MAJOR_CREL 1/2/3/...
- > MCAN Step of Core Release
#define C_ENABLE_MPC5700_MCAN_MAJOR_CREL_STEP 0/1/2/3/...
- > MCAN Sub Step of Core Release
#define C_ENABLE_MPC5700_MCAN_MAJOR_CREL_SSTEP 0/1/2/3/...
- > Non ISO Operation
#define CAN_FD_NISO 0 = ISO 11898-1:2015 / 1 = Bosch CAN FD Spec. V1.0
- > > 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 usage of 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 CAN_NONE/CAN_NOTIFY_ISSUE/CAN_NOTIFY_MAILBOX
- > Use Overrun detection
#define CAN_OVERRUN_NOTIFICATION CAN_NONE/ CAN_DET/ CAN_APPL
- > Select MicroSar version
#define CAN_MICROSAR_VERSION CAN_MSR30/ CAN_MSR40/ CAN_MSR403
- > Tx Cancellation of Identical IDs
#define CAN_IDENTICAL_ID_CANCELLATION STD_ON/STD_OFF

8.2 Link-Time Parameters

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

- > Maximum amount of used controllers and Tx mailboxes (has to be set for post-build variants at link-time)
- > Rx Queue size
- > Controller mapping (mapping of logical channel to hardware node).
- > CAN hardware base address.

8.3 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)
- > CAN hardware Fifo depth
- > CAN hardware clock and bit timing settings

8.4 Configuration with da DaVinci Configurator

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

9 AUTOSAR Standard Compliance

9.1 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 Link Time or Post Build variants. It's recommended that higher layer generator use Values (ObjectIDs) from EcuC file. Vector CAN Interface does so.	3.0.6
	For the acceptance filtering a maximum of 64 filters per CAN channel is supported in case of GENy is used as Generation Tool.	

9.2 Hardware Limitations

8.2.1 Tx side

MCAN Tx Event FIFO is not supported.

MCAN Tx Queue is not supported.

All available buffers per CAN (32) are configured as dedicated Tx buffers.

8.2.2 Rx side

SREQ00014271 "message reception shall use overwrite mode" is not fulfilled for FullCAN messages due to hardware behaviour.

8.2.3 Used resources

Please note that the theoretical possible maximum configuration for the RH850P1xC derivative requires more RAM space in the Shared Message RAM than there is actual available.

For each CAN channel the following elements can be configured. If the required size for a distinct configuration exceeds the maximum available RAM space in hardware then the configuration tool issues an error during generation time and you are requested to tailor down your configuration until it fits into the available Shared Message RAM.

Resource usage for one CAN channel:

Area	Address range	Max size (byte)	Max. number of elements
------	---------------	--------------------	----------------------------

Std Filter	0x0000 – 0x01FF	512	128
Ext Filter	0x0200 – 0x03FF	512	64
Rx FIFO 0	0x0400 – 0x07FF	1024	64
Rx FIFO 1	0x0800 – 0x0BFF	1024	64
Rx Buffer	0x0C00 – 0x0FFF	1024	64
TxEvt FIFO	0x1000 – 0x10FF	256	32
Tx buffer	0x1100 – 0x12FF	512	32
0x1300		4864 bytes total	

Thus a maximum of “4864 * NumberOfChannels” can theoretically be configured but less RAM is physically available. You are requested to reduce the areas according to your needs.

Please note that the “Tx Buffer region” and the “TTCAN region” (for channels with TTCAN support) for each channel is restricted to a dedicated address.

This is not consistent for all hardware releases, please refer to your hardware manufacturer documentation (see ch. 2 “Hardware Overview”).

9.2.1 Initialization of the CAN Message RAM

The internal SRAM features Error Correcting Code (ECC). Because these ECC bits can contain random data after the device is turned on, all SRAM locations must be initialized before being read by application code. Initialization is done by executing 64-bit writes to the entire SRAM block. The value written does not matter at this point, so the Store Multiple Word instruction will be used to write 16 general-purpose registers with each loop iteration.

By default the CAN driver tries to accomplish this initialization. Due to the need of using assembler code notation it might happen that specific options for a distinct compiler (assembler) are not appropriate. If so, you can feel free to disable the CAN driver internal initialization (see below on how to) and use your own initialization instead of.

To disable the CAN driver internal initialization use a “User Config File” containing the following preprocessor definition:

```
#define CAN_ECC_INIT    STD_OFF
```

Put your initialization into execution just before calling Can_Init(). The MCAN clock must be available at this point of time.

Please refer to your hardware manufacturer documentation (see ch. 2 “Hardware Overview”) for the address layout.

9.3 Vector Extensions

Refer to Chapter 4.1 “Features” listed under “**AUTOSAR extensions**”

10 Glossary and Abbreviations

10.1 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 10-1 Glossary

10.2 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 10-2 Abbreviations

11 Contact

Visit our website for more information on

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

www.vector.com