MICROSAR Classic CAN Driver

Technical Reference

ARM32 MCAN
Version 4.11.01

Author	Meid
Status	Released

1 Document Information

1.1 History

Author	Date	Version	Remarks
P. Herrmann	2017-02-22	1.00.00	Creation based on SPC58xx description
P. Herrmann G.Pflügel	2017-04-25	1.01.00	Added latest MCAN Bosch Errata (#16, #17, #18) Added MCAN independent Errata for Aurix Plus
G.Pflügel	2017-07-26	2.00.00	Restructure of history
P. Herrmann	2017-08-03	2.01.00	Updated SPC574Kxx derivative decription for new cut 2.4 hardware revision. Enhanced description in chapter - 4.8.3 “Hardware Loop Check / Timeout Monitoring” - 4.9 “Hardware Specific”
G.Pflügel	2017-08-21	2.02.00	- Platform SAM V71 and Traveo merged together and renamed to platform Arm32Mcan - Platform Telemaco and compiler ARM added to platform Arm32Mcan
P. Herrmann	2017-09-18	2.03.00	Enhanced ch. 4.8.1 “Dev. Error Reporting”
P. Herrmann	2017-10-05	2.04.00	Added Silent Mode
P. Herrmann	2017-11-21	2.05.00	Template update, enhanced Silent Mode description
P. Herrmann	2018-01-15	2.06.00	Dynamic MCAN Revision detection
M. Huse	2018-02-27	2.07.00	Extended Ram Check
P. Herrmann	2018-03-02	2.08.00	Telemaco3P STA1385 Cut2.1
G.Pflügel	2018-03-28	2.09.00	Tricore TC38x and TC39x Step_B added
M. Huse	2018-04-04	2.10.00	BCM89103 added
M. Huse	2018-04-11	2.11.00	Updated API description
C. Huo	2018-04-12	2.12.00	TDA3x added
M. Huse	2018-04-19	3.00.00	Updated document for multi driver compatibility
M. Huse	2018-05-07	3.00.01	Updated ISR section for multi driver compatibility.
G.Pflügel	2018-06-11	3.01.00	HighTec GNU for Tricore
M. Huse	2018-07-23	3.02.00	Visconti5 added
G.Pflügel	2018-09-21	3.03.00	Support HighTec GNU for Spc58xx
M. Huse	2018-10-23	3.04.00	Support IAR compiler for ARM. Traveo2 added
M. Huse	2018-12-17	3.05.00	Updated description for Generic PreTransmit. Added areas for Protected Register Access
M. Huse	2019-02-11	3.06.00	ATSAME5X added. Updated Mcan Errata sheet reference
G.Pflügel	2019-03-21	3.07.00	Tricore TC35x added

P. Herrmann	2019-03-28	4.00.00	R22 update
M. Huse	2019-04-18	4.01.00	ATSAMC21 added. Support ARM6 compiler for ARM derivatives. BCM89107 added. TC37X added. Added TriCore specific hardware loop and errata description.
M. Huse	2019-05-08	4.02.00	Support for Panasonic AS1. Updated feature table.
M. Huse	2019-06-11	4.03.00	Added SPC58EN8x to supported derivatives.
M. Huse	2019-06-22	4.04.00	Support for Traveo2 High
M. Huse P. Herrmann	2019-09-30	4.04.01	Rename TT_CAN_x channels to MCAN_x, GHS support for Tricore
M. Huse G.Pflügel	2019-10-14	4.04.02	Added DET description for Mcan Message RAM access failure (MRAF). Extended list of supported derivatives for Traveo2 Added CanEccInit Application API Description Update description for ApplCanInitPostProcessing Support for Tx Hardware FIFO Added Errata 19 and 20 Added support for Stm32
G.Pflügel	2020-02-17	4.04.03	Tricore TC337 added
G.Pflügel	2020-02-20	4.04.04	TC337 changed into TC33x
M. Huse	2020-02-27	4.04.05	Support for Traveo2 2D Cluster and additional derivatives.
R. Elabed	2020-03-02	4.04.06	Support for SPC58EE80 Fixed Description for FD support for Revision 3.0.x.
G.Pflügel	2020-04-07	4.05.00	Tricore derivatives added
M. Huse N. Jayakrishnan	2020-04-20	4.06.00	Added description for Tricore RAM initialization hardware loop Support for AWR1642 Support for TCC8030
N. Jayakrishnan	2020-05-14	4.06.01	Added MPC5775B/E to supported derivatives
M. Huse	2020-06-05	4.07.00	Support for TI TDA4VM
M. Huse	2020-09-15	4.08.00	Added additional Traveo2 derivative. Added additional STM32 derivative. Added ATSAMV70 to supported derivatives Support for DRA821. Updated protected area section for Tricore. Updated error reporting information.
vishum	2020-10-14	4.08.01	Added support for SPC58xHx – Chorus10M Expanded hardware controller table for Jacinto7
visnaj vishum	2020-11-19	4.08.02	Change in hardware loop description Extended TriCore controller and compiler support

			Added support for Traveo2 derivative CYT3DLx
meid vishum	2021-01-21	4.08.03	Support for IWR68x Improvements of critical section description
meid	2021-02-17	4.08.04	Added support for TPR12, AWR2944 and AWR2943
meid	2021-02-22	4.08.05	Added support for AWR18, IWR16, IWR18 and IWR64
visred vishum	2021-05-10	4.09.00	Support Virtual Addressing Support CYT2B6x
meid	2021-06-16	4.09.01	Support for Stellar Added LlvmDiab and LlvmHighTec
vishnj	2021-08-11	4.10.00	Support Security Event Reporting
visped	2021-10-15	4.10.01	Added LlvmTexasInstruments compiler Added additional reference manual for Jacinto 7 Added Stellar derivatives SR6Px SR6Gx
visped meid	2021-11-10	4.10.02	Updated Tx Mailbox Layout Add support for TDA4VE, TDA4AL, TDA4VL
vishnj	2021-12-23	4.10.03	Support TCC70xx Add a TeleChips TCC section to the Platform specific behavior chapter
vishum	2022-02-21	4.11.00	Added CYT2Cx Added AM273x Enhanced description of filter elements
Meid	2022-04-20	4.11.01	Added Support for AWR68

Table 1-1 Document History

1.2 Reference Documents

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

[6]	M_CAN Controller Area Network Errata Sheet	REL2018 0720
[7]	Appl. Note AN-ISC-8-1190 CAN Self Diag	1.1.0
[8]	ISO DIS 11898-1	2015

Table 1-2 Reference Documents

1.3 Scope of the Document

This document describes the functionality, API and configuration of the MICROSAR Classic 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.

Content

Illustrations

Tables

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
SAMV71 SAMV70	GHS, Keil, ARM, ARM6, GNU, IAR, TI, LlvmDiab, LlvmHighTec, LlvmTexasInstruments	Atmel-44003E-ATARM-SAM V71 SAM-E70-S70-V70-V71-Family-Data-Sheet	Datasheet_ 12-Oct-16 DS60001527D, 2019
SAME51/E54		SAMD5x/E5x Family Data Sheet	2018, 01507B
SAMC21(E/G/J)		Atmel-42365-SAM-C21_Datasheet.pdf	2016-11
S6J31x		S6J3110 Series, 32-bit Microcontroller, FR5 Family, S6J311EJAA/S6J311AHAA, HARDWARE MANUAL	Publication Number S6J3110_MN708-00004, Revision 0.1 Issue Date February 28, 2014
S6J32x		Spansion® Traveo TM Family, 32-BIT MICROCONTROLLER Platform Part HARDWARE MANUAL	Publication Number MN708-00006-1v0-E Rev.1.0, Oct.07,2014
S6J33x		S6J3300 Series, 32-bit Microcontroller, Spansion® Traveo TM Family Hardware Manual	Publication Number S6J3300_MN708-00009 Revision 1.0 Issue Date August 06, 2015
S6J34x		S6J3400 Series, 32-bit Microcontroller, Spansion®, Traveo TM Family	Publication Number S6J3400_SIL15-BE002 Revision 0.02 Issue Date August 31, 2015
STA1375 STA1385		Telemaco3P_RM_Rev1_DraftA_review Telemaco3P_RM_Rev 1.0	1.0_DraftA, DraftB , 11 July 2017
BCM89103 BCM89107		Integrated BroadR-Reach® Camera Endpoint Microcontroller, Technical Reference Manual BCM89103	89103-TRM100-R August 10, 2016
TDA3x TDA4VM TDA4VE TDA4VL TDA4AL DRA829J DRA829V DRA821X		TDA3x_SR2.0_SR1.0A_SR1.0_Public_TRM_vInitial J721E DRA829/TDA4VM/AM752x Processors Silicon Revision 1.0 TDA4AL_VL_VE_TRM_2021_10_29_SPRUJ08__draft J7200 DRA821 Processor Silicon Revision 1.0 TDA4VMDSSR1_0SR1_12021_03_SPRSP36G	SPRUIE7 June 2017 SPRUIL1A November 2019 SPRUJ08 OCT 20 SPRUIU2 March 2020 SPRSP36G – MARCH 2021
TCC8030		FULL SPECIFICATION TCC8030	Rev. 0.02 2018-05-18

Derivative	Compiler	Hardware Manufacturer Document	Version
TCC70xx		FULL SPECIFICATION TCC70xx	Rev. 0.01 2021-07-21
AWR1642 AWR2944 AWR2943 TPR12 AWR18 AWR68 IWR16 IWR18 IWR64 IWR68 AM273x		AWR1642__DS__2018_04__SWRS203A AWR294x_TRM_SPRUIV5_0p4 TPR12__TRM__2020_09_SPRUIU0_v3 AWRxxxx__TRM_2020__04_SWRU520D IWRxxxx__TRM__2020_06__SWRU522E AM273x Technical Reference Manual	SWRS203A – 5/2017 – Rev 4/2018 SPRUIV5 5/2020 SPRUIU0 9/2020 SWRU520D 5/2017– Rev 4/2020 SWRU522E 5/2017– Rev 6/2020 SPRUIU0, 01/2022
TMPV7706XBG		TMPV770 Series Reference Manual SI: CAN FD Interface	Revision 0.1
CYT2B7X CYT2B9X CYT3BBX CYT4BBX CYT4BFX CYT4DNX CYT2BLX CYT3DLx CYT2B6x CYT2CLx CYT2C9x		Traveo™ II Automotive Body Controller Entry Registers (TMR) (002-19567) CYT2B9 Datasheet (002-22825) CYT4BF Datasheet (002-21617) Traveo™ II Automotive Body Controller High Registers Technical Reference Manual (TRM) Traveo™ II Automotive Cluster 2D Family Architecture Technical Reference Manual (TRM) CYT2BL Datasheet (002-28876) CYT3DL Datasheet (002-27763) CYT2B6 Datasheet (002-25756) CYT2CL Datasheet (002-32508)	Rev. **, 06/2017 Rev *B 09/2018 Rev *D, 01/2019 Rev *A 09/2018 Rev. *A 05/2019 Rev. *B 04/2020 Rev. *B 09/2020 Rev. *A 06/2020 Rev. *B 09/2021
MN5460AA0UB		AS1_CANFD_regmap.pdf AS1_CAN_FDC_ext.pdf	Ver 1.00 Ver 1.05
STM32H742 STM32H743 STM32H750 STM32H753 STM32H7A3		STM32H742, STM32H743/753 and STM32H750 Value line advanced Arm®-based 32-bit MCUs STM32H7A3/7B3 and STM32H7B0 Value line advanced Arm®-based 32-bit MCUs	RM0433 Rev.7 02/2020 RM0455 Rev.4 07/2020
SR6Px SR6Gx		SR6P7x 32-bit ARM® Cortex®-R52 architecture microcontroller for automotive ASILD applications	October 2018 RM0459 Rev 1

Table 2-1 Supported Hardware Overview

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

Compiler: List of Compilers the CAN Driver is working with

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

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

3 Introduction

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

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

Supported AUTOSAR Release*:	4	
Supported Configuration Variants:	Pre-Compile, Post-Build Loadable, Post-Build Selectable (MICROSAR Identity Manager)	
Vendor ID:	CAN_30_MCAN_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	CAN_30_MCAN_MODULE_ID	80 decimal (according to ref. [2])
AR Version:	CAN_30_MCAN_AR_RELEASE_MAJOR_VERSION CAN_30_MCAN_AR_RELEASE_MINOR_VERSION CAN_30_MCAN_AR_RELEASE_REVISION_VERSION	AUTOSAR Release version BCD coded
SW Version:	CAN_30_MCAN_SW_MAJOR_VERSION CAN_30_MCAN_SW_MINOR_VERSION CAN_30_MCAN_SW_PATCH_VERSION	MICROSAR Classic 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.

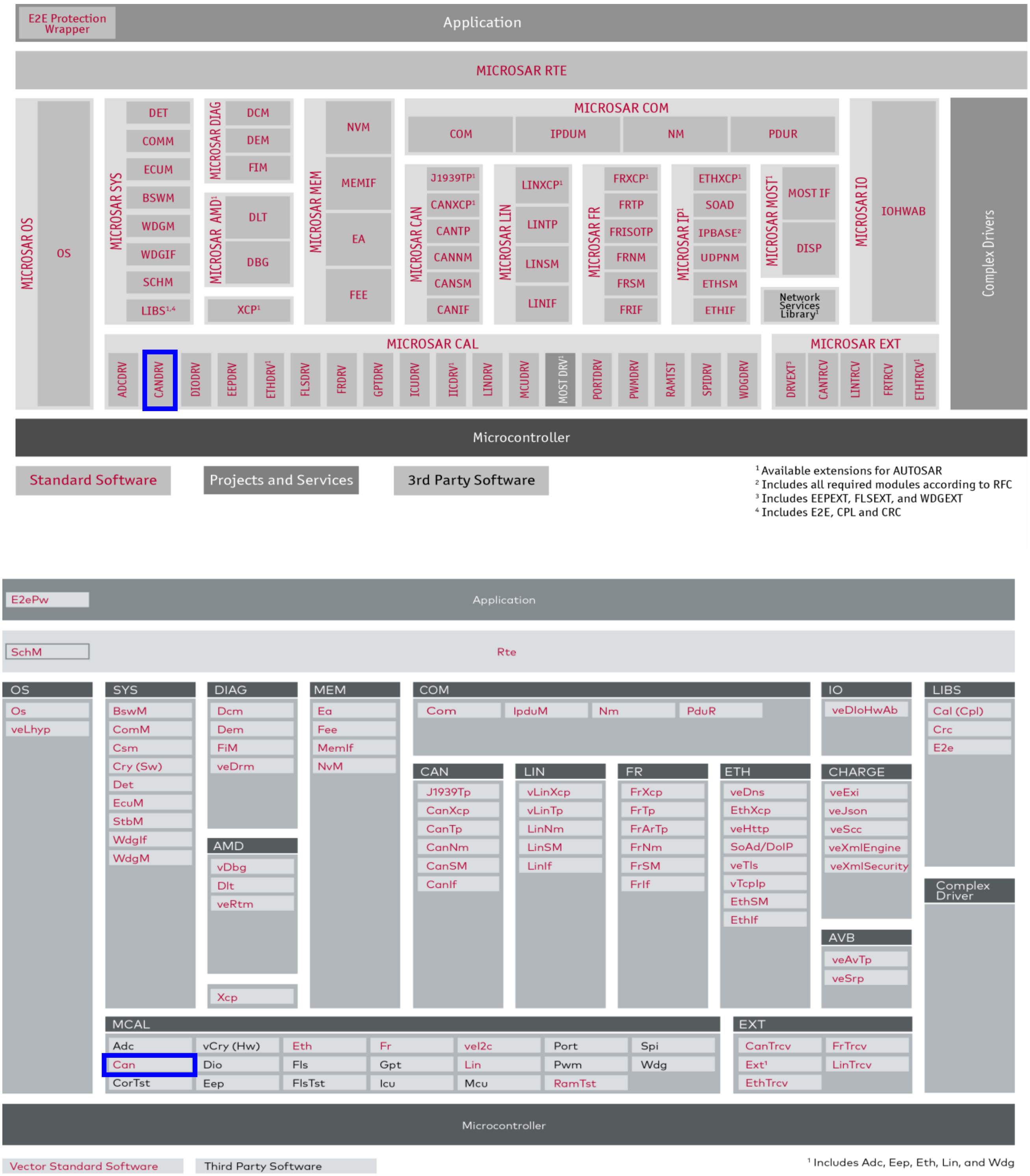

Figure 3-1 AUTOSAR architecture

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

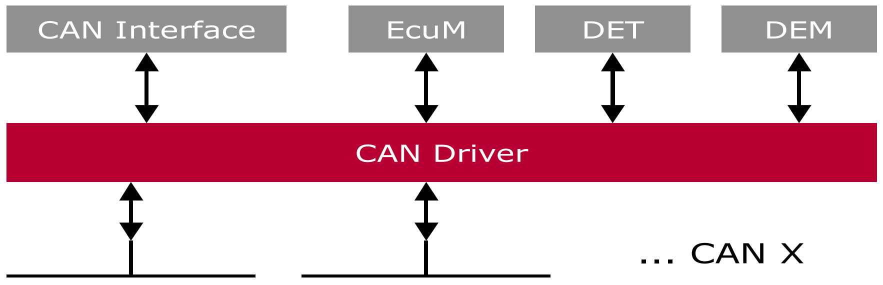

Figure 3-2 Interfaces to adjacent modules of the CAN

4 Functional Description

4.1 Features

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

The "supported" and "not supported" features are presented in the following table.

For further information of not supported features also see chapter 9.

Feature Naming	Short Description	supported
Initialization		
PowerOnInit	General driver initialization function `Can_Init()`	■
DeInit	Support de-initialization routine (since ASR 4.4.0)	□
InitMemory	Support initialization of memory at power on.	■
Communication		
Transmission	Transmitting CAN frames.	■
Transmit confirmation	Callback for successful Transmission.	■
Reception	Receiving CAN frames.	■
Receive indication	Callback for receiving frame.	■
Controller Modes		
Sleep	Controller support `SLEEP` mode (power saving).	□
Wakeup over CAN	Controller support `WAKEUP` over CAN. Limitation: `Can_CheckWakeup()` is not encapsulated by preprocessor switch (since ASR 4.2.0)	□
Stop	Controller support `STOP` mode (passive to CAN bus).	■
Bus Off detection	Callback for `BUSOFF` event.	■
Silent Mode	Support Silent Mode where the controller only listen passive.	■
Wakeup over ICU	Support wakeup over ICU (since ASR 4.2.0).	□
MirrorMode / BusMirroring	Support message mirroring where the controller supports generic confirmation function for mirroring and support an API to activate and deactivate this.	■
Polling Modes		
Tx Confirmation	Support polling mode for Transmit confirmation.	■
Rx Reception	Support polling mode for Reception.	■
Wakeup	Support polling mode for `WAKEUP` event.	□
Bus Off	Support polling mode for `BUSOFF` event.	■
Mode	Support polling mode for mode transition.	■
Mailbox objects		
Tx BasicCAN	Standard mailbox to send CAN frames (Used by CAN Interface data queue).	■

Tx Hardware FIFO	Using a hardware FIFO buffer for a Tx BasicCAN mailbox	■*
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 number of mailboxes.	32
Rx FullCAN	Separate mailbox for special Rx message used.	■
Maximum amount	Available number of mailboxes.	64
Rx BasicCAN	Standard mailbox to receive CAN frames (FIFO 0/1 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.	1 (64) 2 (128)
Symbolic Name Values	Generate Symbolic Name Values for CanHardwareObjects Limitation: “Symbolic Name Values” may change their values after precompile phase so do not use it for Link-time or Post-build variants. (since ASR 4.0.0)	■
Others		
DET	Support Development Error Detection (error notification).	■
Version API	API to read out component version.	■
Maximum supported controllers	Maximum number of supported controllers (hardware channels)	20
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 (BRS; Tx and Rx).	■
CAN FD Mode2	FD frames up to 64 data bytes (FULL support; Tx and Rx).	■
Hardware Loop Check (Timeout monitoring)	To avoid possible endless loops (occur by hardware issue).	■
Multiple CAN driver	API infixed CAN driver support	■
Pretended Networking	Support pretended networking (since ASR 4.2.0)	☐
Individual Polling	Support individual polling mode (selectable for each mailbox separate). Limitation: AutoSar feature multiple cyclic functions with different time periods are not supported. The polling take part in the same main function calls (since ASR 4.2.0).	■*
TriggerTransmit	Support trigger transmit (since ASR 4.2.0)	■
EcuC partition map	Support EcuC partition mapping (since ASR 4.4.0).	■

Security Event Reporting	Support security event reporting (since ASR 20-11)	■
AutoSar extensions		
Multiple Rx Basic CAN	Support Multiple Rx BasicCAN objects. This gives the possibility to use additionally Fifo-1 with 64 additional elements. Overruns can be avoided by optimizing the acceptance filtering.	■*
Multiple Tx Basic CAN	Support Multiple Tx BasicCAN objects. Used to send different Tx groups over separate mailboxes with different buffering behavior (see CanInterface). (“Cancellation of Tx objects” is not possible with this feature activated)	■*
Rx Queue	Support Rx Queue. This gives the possibility to buffer received data in interrupt context but handle it asynchronous in polling task.	■*
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. Nested means that higher prior interrupts may appear in the already active CAN interrupt context.	■
Support Mixed ID	Force CAN Driver to handle mixed ID (standard and extended ID) at pre-compile-time to expand the ID type later.	■
Optimize for one controller	Activate this for 1 controller systems when you never will expand to multi-controller. So that the CAN Driver works more efficient	■
Size of Hw HandleType	Support 8bit or 16bit Hardware Handles depend on hardware usage.	■
Generic PreCopy	Support a callback function for receiving any CAN message (following callbacks could be suppressed) Limitation: AutoSar feature name is LPDU callback function this is not fully supported because the API signature is different (since ASR 4.2.0)	■
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()`, `Can_GetControllerStatus()`, `Can_GetControllerMode()`, `CanGetControllerRxErrorCounter()`, `CanGetControllerTxErrorCounter()`)	■
Interrupt Category selection	Support Category 1 or Category 2 Interrupt Service Routines for OS	■
Overrun Notification	Support DET or application notification cause by `OVERRUN` (`OVERWRITE`) of an Rx message (BasicCAN and FullCAN) Please note that ‘Overrun’ is supported for BasicCAN objects but is not available for FullCAN objects. 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 - another FullCAN object, or - a BasicCAN object, or - the message may be rejected, depending on the filter configuration. Limitation: overrun will be only notified when DET is activated (since ASR 4.4.0 the overrun will be also notified without DET)	
RAM Check	Support CAN mailbox RAM check	■
Extended RAM Check	Support extended RAM check. Handling of individual deactivated mailboxes and controllers.	■
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 right before transmission)	■
Integrity OS	Support of Integrity OS (virtual addressing).	■

Table 4-1 Supported features

■ Feature is supported

☐ Feature is not supported

* HighEnd Licence only

** Project specific (may not be available)

4.2 Initialization

Can_30_Mcan_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_30_Mcan_InitController().

MICROSAR401 only: baud rate settings given by Can_InitController parameter.

Can_30_Mcan_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_30_Mcan_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_30_Mcan_Init() and Can_30_Mcan_InitController() have to be called again. It is recommended to use this function before calling Can_30_Mcan_Init() to secure that no startup-code specific pre-initialized variables affect the driver startup behavior.

4.3 Communication

Can_30_Mcan_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 into the mailbox buffer the ID, DLC and data may be changed by `Appl_30_Mcan_GenericPreTransmit()` callback.

When “Generic Confirmation“ is activated the callback Appl_30_Mcan_GenericConfirmation() will be called before CanIf_TxConfirmation() and the call to this can be suppressed by Appl_30_Mcan_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_30_Mcan_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_30_Mcan_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 … 32	There are a total amount of 32 Tx Buffers available. These are shared between the Tx FullCANs and Tx BasicCANs. Each Tx FullCAN object is used to transmit message with specific ID and requires 1 Tx Buffer. 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 Tx FullCAN hardware objects.

(N+1) … M	**Tx BasicCAN**	0 … 32	There are a total amount of 32 Tx Buffers available. These are shared between the Tx FullCANs and Tx BasicCANs. All Tx BasicCan objects transmit any message IDs. Tx BasicCANs objects are split in 3 types: 1) Tx BasicCAN This type of object requres 1 Tx Buffer. 2) Tx BasicCAN MUX(Multiplexed Transmission) This type of requres 3 Tx Buffers. This will send the lowest ID first. 3) Tx BasicCAN Hw FIFO This type of requres 2 to 32 Tx Buffers. The amount of Tx Buffers is defined by the user using “Object Hw size” parameter. The messages will be sent in FIFO sequence. Only one (1) Tx BasicCAN can be configured with Tx Hw FIFO. Examle: 5 x Tx FullCAN + 3 x Tx BasicCAN + 3 x Tx BasicCAN MUX + 1 x Tx BasicCAN Hw FIFO(with 10 buffers) Tx FullCAN = 5 x 1 = 5 Tx Buffers Tx BasicCAN = (3 x 1) + (3 x 3) + (1 x 10) = 22 Tx Buffers ⇨ 27 used Tx Buffers ⇨ 5 unused Tx Buffers If the transmit message object is busy, the transmit requests are stored in the CAN Interface queue (if activated).
(M+1) …P	**Rx FullCAN**	0 … 64	There are a total amount of 64 dedicated Rx Buffers available. These are used as Rx FullCANs. Each Rx FullCAN object is used to receive message with specific ID and requires 1 dedicated Rx Buffer. Each Rx FullCAN object will consume 1 filter element depending on the ID type (Standard/Extended). The total amount of available filters are shared with the Rx BasicCAN objects. 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.

P+1	**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 the Rx FullCAN objects and FIFO-1, if available.
P+2	**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 the Rx FullCAN objects and FIFO-0.

The “CanObjectId” (ECUc parameter) numbering is done in following order: Tx FullCAN, Tx BasicCAN, Rx FullCAN, 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	Message ID Low to High
Tx BasicCAN	Message ID Low to High or FIFO
Rx FullCAN	Message 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 method mentioned above.

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. The filters are shared between the Rx FullCAN objects and the Rx BasicCAN (FIFO-0/1) objects.

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_30_Mcan_SetControllerMode(). The last requested transition will be executed. The upper layer has to take care about valid transitions.

The following mode changes are supported:

`CAN_T_START`

`CAN_T_STOP`

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 Power Down Mode

The CAN controller does not support a Sleep/Wakeup Mode, nevertheless power saving is possible with the “Power Down” Mode via a Clock Stop Request (CSR).

After requesting Clock Stop all pending transmissions have to be completed then the CAN Controller waits until bus idle state is detected. Then the CAN Controller sets Initialization to one to prevent any further CAN transfers. Now the CAN Controller acknowledges that it is ready for power down by setting Clock Stop Acknowledge. At this point of time the CAN Controller clock inputs may be switched off.

To leave Power Down Mode, the application has to turn on the CAN Controller clocks before resetting Clock Stop Request. The CAN Controller will acknowledge this by resetting Clock Stop Acknowledge. Afterwards the CAN communication can be restarted by resetting the initialization mode.

The application is, if configured, requested to turn off the clocks for CAN and Host controllers. When the Clock Stop Request returns, then it is assumed that the clocks are off.

In the same way the application is requested to turn on the CAN clocks during power up before the CAN starts communication. When the Clock Start Request returns, then it is assumed that the clocks are on.

Please note that the user callback function that will be called in case of a clock stop request acknowledge must be defined via a user-configuration file (see example below).

Example for a user – configuration file entry defining the Clock Start/Stop callback functions:

```
#define Appl_30_McanCanClockStop(CanChannel)
ApplCanClockStopAcknowledged(CanChannel)
/* will be called when the application is allowed to turn off the clocks for CAN and Host */
#define Appl_30_McanCanClockStart(CanChannel)
ApplCanClockStartRequested(CanChannel)
/* will be called when the application must turn on the clocks for CAN and Host before communication is
started */
```

The parameter “`CanChannel`” is either of type “void” in case of a single channel configuration or it contains the number of the CAN channel in case of a multi channel configuration.

4.4.4 Bus Off

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

4.4.5 Silent Mode

Support API (`Can_30_Mcan_SetSilentMode()`) to switch into ‘SilentMode’ where the controller does not take part on BUS communication (no ACK) but can listen for messages. Please refer also to ISO 11898 bus monitoring.

The MCAN describes this mode as Bus Monitoring Mode:

In Bus Monitoring Mode (see ISO 11898-1:2015, 10.14 Bus monitoring), the M_CAN is able to receive valid data frames and valid remote frames but cannot start a transmission. In this mode, it sends only recessive bits on the CAN bus. If the M_CAN is required to send a dominant bit (ACK bit, overload flag, active error flag), the bit is rerouted internally so that the M_CAN monitors this dominant bit, although the CAN bus may remain in recessive state. In Bus Monitoring Mode register TXBRP is held in reset state.

The Bus Monitoring Mode can be used to analyze the traffic on a CAN bus without affecting it by the transmission of dominant bits.

In case of an error condition or overload condition no dominant bits are sent, instead the MCAN waits for the occurrence of bus idle condition to resynchronize itself to the CAN communication. The error counters (ECR.REC, ECR.TEC) are frozen while Error Logging (ECR.CEL) is active. This can be used in applications that adapt themselves to different CAN bit rates.

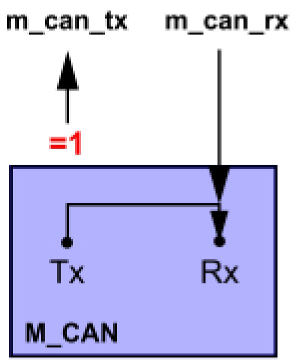

Figure 4-1 Bus Monitoring Mode.

Caution
With activated “Silent Mode” do not use any other API than
Can_30_Mcan_SetSilentMode(“CAN_SILENT_INACTIVE”),
Can_30_Mcan_SetControllerMode(“START” or “STOP”) or
Can_30_Mcan_ChangeBaudrate() or Can_30_Mcan_SetBaudrate().
Especially do NOT request any transmission.

4.4.6 Dynamic MCAN detection

In the case of several platform hardware versions with different MCAN Revisions are used, the integrated MCAN Revision can be detected during runtime by the CAN Driver. If so, the CAN Driver adapts itself to the underlying MCAN Revision.

To enable this mode the preprocessor switch “`C_30_MCAN_ENABLE_DYNAMIC_MCAN_REVISION`” must be defined via a user configuration file.

Adaptations which do not need additional data are accomplished internally now. For adaptations which need additional data the user callback function "`Appl_30_McanCanInitPostProcessing()`" has to be enabled in addition. This callback function is called during initialization time of the MCAN and thus allows to overwrite MCAN registers with values which are MCAN Revision dependent (see ch. 7.2.44).

Caution
Please note that the dynamic MCAN detection only works upwards.
This means that the configuration is always based on MCAN Revision 3.0.x.
The effective underlying MCAN Revision may be either 3.0.x or 3.2.x.

4.5 Re-Initialization

A call to `Can_30_Mcan_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_30_Mcan_DisableControllerInterrupts()` and `Can_30_Mcan_EnableControllerInterrupts()` are used to disable and enable the controller specific Interrupt, Rx, Tx, Wakeup and BusOff (/ Status) together. These functions can be called nested.

4.7 Main Functions

`Can_30_Mcan_MainFunction_Write()`, `Can_30_Mcan_MainFunction_Read()`, `Can_30_Mcan_MainFunction_BusOff()` and `Can_30_Mcan_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_30_Mcan_MainFunction_Write()`" and "`Can_30_Mcan_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_30_Mcan_MainFunction_Read()`" context.

Can_30_Mcan_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_30_MCAN_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_30_MCAN_E_PARAM_POINTER	API gets an illegal pointer as parameter.
CAN_30_MCAN_E_PARAM_HANDLE	API gets an illegal handle as parameter
CAN_30_MCAN_E_PARAM_DLC	API gets an illegal DLC as parameter
CAN_30_MCAN_E_PARAM_CONTROLLER	API gets an illegal controller as parameter
CAN_30_MCAN_E_UNINIT	Driver API is used but not initialized
CAN_30_MCAN_E_TRANSITION	Transition for mode change is illegal
CAN_30_MCAN_E_DATALOST (value: 0x07, AutoSar extension)	Rx overrun (overwrite) detected
CAN_30_MCAN_E_PARAM_BAUDRATE (value: 0x08, AutoSar extension)	Selected Baudrate is not valid
CAN_30_MCAN_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_30_MCAN_E_TIMEOUT_DET (value: 0x11, AutoSar extension)	Same as CAN_30_MCAN_E_TIMEOUT for DEM but this is notified to DET due to switch "CAN_30_MCAN_DEV_TIMEOUT_DETECT" is set to STD_ON (see configuration options)
CAN_30_MCAN_E_GENDATA (value:0x12, AutoSar extension)	Standardized issue for inconsistent generated data
CAN_30_MCAN_E_MRAF (value: 0x13, AutoSar extension)	Mcan Message RAM access failure occurred.
kCanErrorMcanRevision (value:0xA2, AutoSar extension)	The configured Mcan Revision is not equal to the Mcan Revision read directly from the underlying hardware during startup.
kCanErrorMcanMessageRAMOverflow (value:0xA3, AutoSar extension)	The address used for a Message RAM access is behind the end address of the available Message RAM.

kCanErrorChannelHdlTooLarge (value:0xA4, AutoSar extension)	The handle used for the channel parameter is larger than the number of configured channels
kCanErrorSICANFDKeyRejected (value:0xA5, AutoSar extension)	Only for Visconti5 (Toshiba platform): Fault injection setting key status is not done in status of key code register.
kCanErrorPwdRejected (value:0xA8, AutoSar extension)	The write protection on the CAN Config registers did not work. The Message RAM address maybe configured wrong and may result in arbitrary communication.

Table 4-2 Errors reported to DET

API from which the errors are reported to DET:	
API ID	**Functions using that ID**
CAN_30_MCAN_VERSION_ID	Can_30_Mcan_GetVersionInfo()
CAN_30_MCAN_INIT_ID	Can_30_Mcan_Init()
CAN_30_MCAN_INITCTR_ID	Can_30_Mcan_InitController()
CAN_30_MCAN_SETCTR_ID	Can_30_Mcan_SetControllerMode()
CAN_30_MCAN_DIINT_ID	Can_30_Mcan_DisableControllerInterrupts()
CAN_30_MCAN_ENINT_ID	Can_30_Mcan_EnableControllerInterrupts()
CAN_30_MCAN_WRITE_ID	Can_30_Mcan_Write(), Can_30_Mcan_CancelTx()
CAN_30_MCAN_TXCNF_ID	CanHL_30_Mcan_TxConfirmation()
CAN_30_MCAN_RXINDI_ID	Can_30_McanBasicCanMsgReceived(), Can_30_McanFullCanMsgReceived()
CAN_30_MCAN_CTRBUSOFF_ID	CanHL_30_Mcan_ErrorHandling()
CAN_30_MCAN_CKWAKEUP_ID	CanHL_30_Mcan_WakeUpHandling(), Can_30_Mcan_Cbk_CheckWakeup()
CAN_30_MCAN_MAINFCT_WRITE_ID	Can_30_Mcan_MainFunction_Write()
CAN_30_MCAN_MAINFCT_READ_ID	Can_30_Mcan_MainFunction_Read()
CAN_30_MCAN_MAINFCT_BO_ID	Can_30_Mcan_MainFunction_BusOff()
CAN_30_MCAN_MAINFCT_WU_ID	Can_30_Mcan_MainFunction_Wakeup()
CAN_30_MCAN_MAINFCT_MODE_ID	Can_30_Mcan_MainFunction_Mode()
CAN_30_MCAN_CHANGE_BR_ID	Can_30_Mcan_ChangeBaudrate()
CAN_30_MCAN_CHECK_BR_ID	Can_30_Mcan_CheckBaudrate()
CAN_30_MCAN_SET_BR_ID	Can_30_Mcan_SetBaudrate()
CAN_30_MCAN_HW_ACCESS_ID (value: 0x20, AUTOSAR extension)	Used when hardware is accessed (call context may vary)

Table 4-3 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_30_MCAN_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 Classic 4.x) or an Application call (*Appl_30_Mcan_CanOverrun*()).

Please note that 'Overrun' is supported for BasicCAN objects but is not available for FullCAN objects.

While the received message is still in the Rx buffer contained (New Data flag is set) for a specific FullCAN object a Message ID Filter Element referencing this specific object will not match, causing the acceptance filtering to continue. Following Message ID Filter Elements may cause the received message to be stored into another Rx Buffer, or into an Rx FIFO, or the message may be rejected, depending on filter configuration.

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_30_MCAN_PROD_ERROR_DETECT == STD_ON`.

The table below shows the Event ID and Event Status given as parameter for calling the DEM. This callout may occur in the context of different API calls (see Chapter "Hardware Loop Check / Timeout Monitoring").

Event ID	Event Status	Short Description
CAN_30_MCAN_E_TIMEOUT	DEM_EVENT_STATUS_FAILED	Timeout in "Hardware Loop Check" occurred, hardware has to be checked or timeout is too short.

Table 4-4 Errors reported to DEM

4.8.3 Hardware Loop Check / Timeout Monitoring

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

Since AUTOSAR4, a synchronous part of mode transitions will be also limited by this timeout mechanism which is no issue but a timing limit. The following asynchronous part of mode transition is handled without Hardware Loop Check.

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

Nevertheless, refer to "short description" below (there may be activities that should be initiated by the application like a reset of the CAN controller or some special mode transitions). If so, the "Hardware Loop Check by Application" is recommended to be used to handle the concerned loop explicitly.

4.8.3.1 Critical Loops

A loop exception must be handled by application like described below.

Loop Name / source	Short Description
kCan_30_McanLoopInit	This is a channel dependent loop called during channel initialization and mode transistion. It is required to account for the delay in update of status register bits caused due to the synchronization mechanism between the CAN clock and Host clock domains. The duration of this loop is expected to be one message length long, as mode transitions occur only when the CAN bus is idle. If the loop cancels, try to reinitialize the controller or reset the hardware.
kCan_30_McanStmFdLoopInit	Only used for the STM32 Can driver: This loop is identical to kCan_30_McanLoopInit, however, only called during the power-on initialization. In addition, the channel parameter should not be considered when this loop is called. The loop will be processed if the first hardware controller does not enter configuration mode.
kCan_30_McanLoopClockStop	This is a channel dependent loop called when Clock Stop is requested in Errata handling. (Please see also ch. 4.4.3 Power Down Mode) The duration of this loop is expected to be one message length long, as clock stop is acknowledged only when the CAN bus is idle. If the loop cancels, try to reinitialize the controller or reset the hardware.

Table 4-5 Hardware Loop Check (critical)

4.8.3.2 Uncritical Loops

No additional application handling needed after loop break.

Loop Name / source	Short Description
kCan_30_McanLoopRxFifo	This channel dependent loop is called in the CanRx Handling. It 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 then, in case of an interrupt driven configuration, the remaining messages in the Fifo(s) will be read not till the next Rx interrupt appears. In case of a polling configuration the polling will continue as usual with the next task cycle.

Table 4-6 Hardware Loop Check (uncritical)

Driver handles the mode transition in an asynchronous way after the synchronous mode transition, so no additional handling is necessary.

Loop Name / source	Short Description
kCan_30_McanLoopMode	Used for short time mode transition blocking (short synchronous timeout). Use for mode changes START and STOP. No issue when timeout occurs.

Table 4-7 Hardware Loop Check (synchronous mode transition)

4.8.4 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_30_Mcan_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_30_Mcan_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_30_Mcan_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_30_Mcan_Init().

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

4.8.5 Extended RAM Check

The CAN Driver supports a check for all accessible CAN Controller's control registers and mailbox registers. The extended RAM check will be executed during power on initialization and by direct call. Mailboxes will be deactivated when pattern check fails or configured values are corrupt. The CAN Controller will be deactivated when at least one mailbox is corrupt or one or more controller register failed the pattern check or configured values are corrupt.

Mailbox and controller stay deactivated until explicitly re-activated. The application is fully responsible to handle this (see Application Note [7] for further information).

API to execute extended RAM check:

```
Can_30_Mcan_RamCheckExecute()
```

Callouts to notify corrupt mailboxes or controllers:

```
CanIf_30_Mcan_RamCheckCorruptController(),CanIf_30_Mcan_RamCheckCor
ruptMailbox()
```

API to re-activate the mailbox or controller again:

```
Can_30_Mcan_RamCheckEnableMailbox(),Can_30_Mcan_RamCheckEnableContr
oller()
```

Please note that only the registers that have both read and write functionality are checked.

4.8.6 Security Event Reporting

The CAN driver supports the detection and reporting of security events observed by the CAN controller. The reporting will be executed during the driver error handling, either in interrupt, or polling context. The associated context data will be reported to the CanIf when a security event is detected.

The security event reporting feature can be activated through the parameter CanEnableSecurityEventReporting in the generation tool.

Three types of error reporting are provided with the security event reporting feature.

4.8.6.1 Error state passive

Reported when the CAN driver detects a transition to error state passive. This is reported when calling the API `CanHL_30_Mcan_ControllerErrorStatePassive()`, which forwards the associated context data to the CanIf API `CanIf_ControllerErrorStatePassive()` during the error handling.

4.8.6.2 Error state bus off

Reported when the CAN driver detects a bus off event. The driver calls the CanIf API `CanIf CanHL_30_Mcan_ControllerBusOff` during the error handling when a bus off event is detected.

4.8.6.3 Error reporting (error types)

Specific CAN error types are reported to the `CanIf API CanIf_ErrorNotification()` from the CAN driver API `CanHL_30_Mcan_ErrorNotification()` during the error handling.

The table below listed the supported CAN error security events:

Security Event	Short Description
Error bit monitoring 1	Reported when a “0” was transmitted and a “1” was read back
Error bit monitoring 0	Reported when a “1” was transmitted and a “0” was read back
Error acknowledge	Reported when acknowledgement check failed
Error form	Reported when a violation of the frame format is detected
Error stuffing	Reported when the stuffing bits is not as expected
Error CRC	Reported when the CRC check failed

Table 4-8 Security Events

4.8.6.4 MCAN Security Event Reporting Restrictions

- A read access to the MCAN protocol status register (PSR) clears all error events listed in the table above. Besides the Security Event Reporting, the API functions: `Can_30_Mcan_GetStatus()` and `Can_30_Mcan_GetControllerErrorState()` read the protocol status register as well.

  Therefore, using these functions in combination with the Security Event Reporting may lead to race condition and the errors to be reported could be lost.

> **Caution**
> Using Security Event Reporting together with `Can_30_Mcan_GetStatus()` or `Can_30_Mcan_GetControllerErrorState()` may lead to race condition so that errors are not reported with Security event reporting.

- Regarding error state passive, a fast transition from error passive to error active then again to error passive state may be reported only once: when the Hardware counters are just around the error passive state, the transition between error passive and error active state could be faster than the software processing time especially in case of polling mode, and so fast transitions could be reported once.

4.9 Hardware Specific

For a correct operation the driver expects all of its registers and the MCAN Message RAM to be accessible in “User Mode”. Please check the Hardware Reference Manual (see chapter 2) for the appropriate measures to be taken like register and memory protection mechanisms.

For a correct operation the driver also expects the correct configuration of interrupt control registers and correct transceiver configuration.

Additionally the clock supply has to be provided and finally it is necessary to configure the port pins correctly to get CAN communication.

> **Please note**
> This configuration work is not part of the CAN Driver.

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.

> **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 hardware features

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).

- Tx Event FIFO is not used
- Tx Queue is not used
- The filtering of High Priority messages are not supported.
- Range Filters are not supported
- Transmit Cancellation is (no longer) supported

4.9.3 Platform specific behavior

4.9.3.1 ATSAMC21

Compare to the Document

Atmel SAM C21E/G/J Datasheet,

Atmel-42365K-SAM-C21_Datasheet_Complete-11/2016,

see chapter 35.8.3. Message RAM Configuration.

The register “MRCFG” is not configurable. Thus, the reset value “0x00000002” is used what means:

2) Behavior of the CAN during standby Sleep Mode:

The CAN GCLK request is always disabled during sleep to conserve power consumption.

b) Memory priority access during the Message RAM read/write data operation:

MEDIUM Sensitive latency

4.9.3.2 Panasonic AS1

For Panasonic AS1 platform the interrupt control must be handled by the application or by OS, please see chapter 6.2.2 and 6.2.3.

The interrupt disabling by driver or in the external interrupt controller is not enough as the interrupt will still be notified to the CPU although it is disabled. Therefore, the Can interrupt enabling/disabling should be handled directly in the Nested Vectored Interrupt Controller.

4.9.3.3 TeleChips TCC

The start address of the CAN message RAM must be configured according to the CAN message RAM start address described in hardware reference manual.

The parameter is defined in CFG5 under: Can/CanGeneral/CanMessageRAM

4.9.4 MCAN specific behavior

Please note that MCAN Revision 3.1.x is the very first one which supports CAN-FD functionality (bitrate switching and full 64 data bytes) on a “per message” basis.

CAN-FD is only supported by the driver in the Revision 3.1.x and higher.

4.10 Virtual Addressing

Virtual Addressing is intended to be used in combination with operating systems that use virtual addresses instead of physical addresses to access the MCAN hardware. The driver sets the virtual base address by calling the application callback function

Appl_30_McanCanPowerOnGetBaseAddress() during Can_30_Mcan_Init(). This function needs to be declared and defined by the application. It is described in section 7.2.46.

> **Please note**
> To enable the Virtual Addressing feature, the user must activate it from Cfg5 to generate:
> #define C_30_MCAN_ENABLE_UPDATE_BASE_ADDRESS

5 Integration

This chapter gives necessary information for the integration of the MICROSAR Classic 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_30_Mcan_Local.h	This is an internal header file which should not be included outside this module
(+) Can_30_Mcan.c	This is the source file of the CAN. It contains the implementation of CAN module functionality.
(+) Can_30_Mcan.lib	This is the library build out of Can_30_Mcan.c, Can_30_Mcan.h and Can_30_Mcan_Local.h
Can_30_Mcan.h	This is the header file of the CAN module (include API declaration)
Can_30_Mcan_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.

File Name	Description
Can_30_Mcan_Cfg.h	Generated header file, contains some type, prototype and pre-compile settings
Can_30_Mcan_Lcfg.c	Generated file contains link time settings.
Can_30_Mcan_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

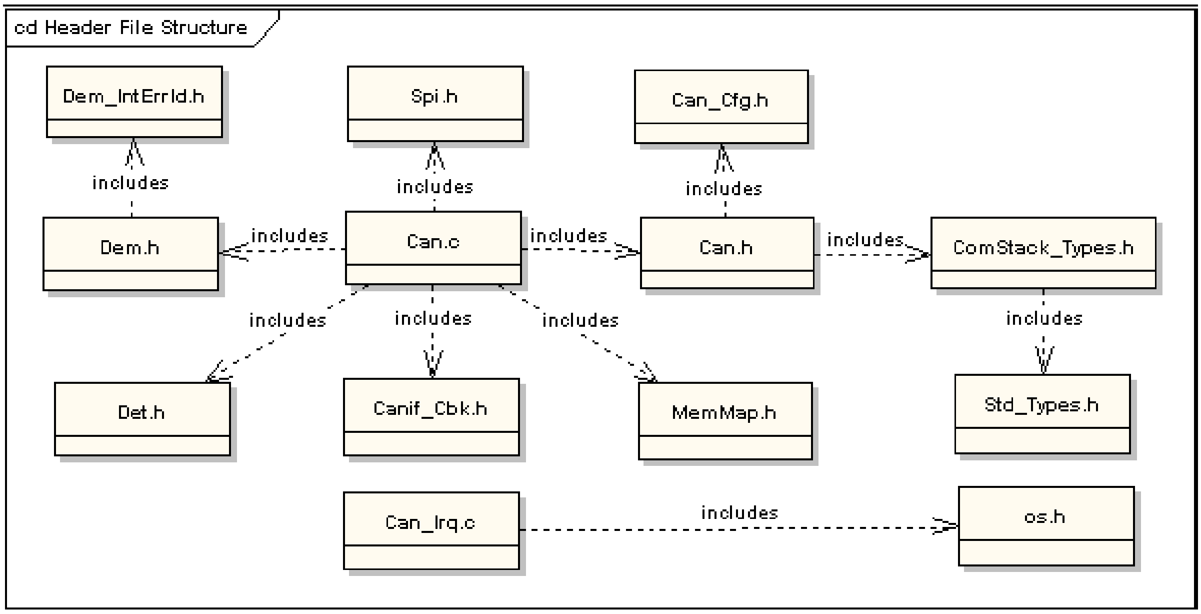

Figure 5-1 Include Structure (AUTOSAR)

Deviation from AUTOSAR specification:

- Additionally the EcuM_Cbk.h is included by Can_30_Mcan_Cfg.h (needed for wakeup notification API).
- ComStack_Types.h included by Can_30_Mcan_Cfg.h, because the specified types have to be known in generated data as well.
- Os.h will be included by Can_30_Mcan_Cfg.h because of used data-types
- Spi.h is not yet used.
- MICROSAR403 only: Can_GeneralTypes.h will be included by Can_30_Mcan_Cfg.h not by Can_30_Mcan.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_30_MCAN_EXCLUSIVE_AREA_0	CanNestedGlobalInterruptDisable/Restore() is used within Can_30_Mcan_MainFunction_Write() and inside the transmit confirmation to assure that transmit confirmations do not conflict with further transmit requests. > Duration is short. > No API call of other BSW inside.
CAN_30_MCAN_EXCLUSIVE_AREA_1	Used inside Can_30_Mcan_DisableControllerInterrupts() and Can_30_Mcan_EnableControllerInterrupts() to secure Interrupt counters for nested calls. > Duration is short. > No API call of other BSW inside. > Disable global interrupts – or – Empty in case Can_30_Mcan_Disable/EnableControllerInterrupts() are called within context of higher or equal priority than the CAN interrupts, and Can_30_Mcan_DisableControllerInterrupts()/Can_30_Mcan_EnableControllerInterrupts() are not called nested.
CAN_30_MCAN_EXCLUSIVE_AREA_2	Used inside Can_30_Mcan_Write() to secure software states of transmit objects. > 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 reentrant.
CAN_30_MCAN_EXCLUSIVE_AREA_3	Used inside Tx confirmation to secure state of transmit object in case of cancellation. (Only used when Vector Interface Version smaller 4.10 used) > 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_30_Mcan_Write() within.
CAN_30_MCAN_EXCLUSIVE_AREA_4	Used inside received data handling (Rx Queue treatment) to secure Rx Queue counter and data. > Duration is short. > No API call of other BSW inside. > Disable Global Interrupts – or – Disable all CAN interrupts.

CAN_30_MCAN_EXCLUSIVE_AREA_5	Used inside wakeup handling to secure state transition. (Only in wakeup Polling Mode) > Duration is short. > Call to DET inside. > Disable global interrupts (do not use CAN interrupt locks here)
CAN_30_MCAN_EXCLUSIVE_AREA_6	Used inside Can_30_Mcan_SetControllerMode() and BusOff to secure state transition. > Duration is medium. > No API call of other BSW inside. > Use CAN interrupt locks here, when the API for one controller is not called in a context higher than the CAN interrupt or Disable global interrupts
CAN_30_MCAN_EXCLUSIVE_AREA_7	Used inside received data handling (Tx hardware FIFO treatment) to secure Tx hardware FIFO counter and data. > Duration is SHORT, modify queue counter and copy data to queue. > No API call of other BSW inside. > Disable Global Interrupts – or – Disable all CAN 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 / *Memory Mapping Sections*	CAN_CODE	CAN_STATIC_CODE	CAN_CONST	CAN_CONST_PBCFG	CAN_VAR_NOINIT	CAN_VAR_INIT	CAN_VAR_PBCFG	CAN_INT_CTRL	CAN_REG_CANCELL	CAN_RX_TX_DATA	CAN_APPL_CODE	CAN_APPL_CONST	CAN_APPL_VAR
CAN_30_MCAN_START_SEC_CODE CAN_30_MCAN_STOP_SEC_CODE	■												
CAN_30_MCAN_START_SEC_STATIC_CODE CAN_30_MCAN_STOP_SEC_STATIC_CODE		■											
CAN_30_MCAN_START_SEC_CONST_8BIT CAN_30_MCAN_STOP_SEC_CONST_8BIT			■										
CAN_30_MCAN_START_SEC_CONST_16BIT CAN_30_MCAN_STOP_SEC_CONST_16BIT			■										
CAN_30_MCAN_START_SEC_CONST_32BIT CAN_30_MCAN_STOP_SEC_CONST_32BIT			■										
CAN_30_MCAN_START_SEC_CONST_UNSPECIFIED CAN_30_MCAN_STOP_SEC_CONST_UNSPECIFIED			■										
CAN_30_MCAN_START_SEC_PBCFG CAN_30_MCAN_STOP_SEC_PBCFG				■									
CAN_30_MCAN_START_SEC_PBCFG_ROOT CAN_30_MCAN_STOP_SEC_PBCFG_ROOT				■									
CAN_30_MCAN_START_SEC_VAR_NOINIT_UNSPECIFIED CAN_30_MCAN_STOP_SEC_VAR_NOINIT_UNSPECIFIED					■								
CAN_30_MCAN_START_SEC_VAR_INIT_UNSPECIFIED CAN_30_MCAN_STOP_SEC_VAR_INIT_UNSPECIFIED						■							
CAN_30_MCAN_START_SEC_VAR_PBCFG CAN_30_MCAN_STOP_SEC_VAR_PBCFG							■						
CAN_30_MCAN_START_SEC_CODE_APPL CAN_30_MCAN_STOP_SEC_CODE_APPL											■		

Table 5-4 Compiler abstraction and memory mapping

The Compiler Abstraction Definitions CAN_30_MCAN_APPL_CODE, CAN_30_MCAN_APPL_VAR and CAN_30_MCAN_APPL_CONST are used to address code, variables and constants which are declared by other modules and used by the CAN Driver.

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

CAN_CODE: used for CAN module code.

CAN_STATIC_CODE: used for CAN module local code.

CAN_CONST: used for CAN module constants.

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

CAN_VAR_*: used for CAN module variables.

CAN_INT_CTRL: is used to access the CAN interrupt controls.

CAN_REG_CANCELL: is used to access the CAN cell itself.

CAN_RX_TX_DATA: access to CAN Data buffers.

CAN_APPL_*: access to higher layers.

6 Hardware Specific Hints

6.1 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 according to your configuration.

As an example the following table shows the corresponding assignment for the derivatives S6J33xx.

Interrupt Functions	
MCAN_0 BaseAddress: 0xB490 0000	Can_30_McanIsr_0
MCAN_1 BaseAddress: 0xB491 0000	Can_30_McanIsr_1
MCAN_2 BaseAddress: 0xB492 0000	Can_30_McanIsr_2
MCAN_3 BaseAddress: 0xB493 0000	Can_30_McanIsr_3
MCAN_4 BaseAddress: 0xB494 0000	Can_30_McanIsr_4
MCAN_5 BaseAddress: 0xB06C 0000	Can_30_McanIsr_5
MCAN_6 BaseAddress: 0xB06D 0000	Can_30_McanIsr_6
MCAN_7 BaseAddress: 0xB06E 0000	Can_30_McanIsr_7

Table 6-1 Hardware Controller – Interrupt Functions

The following table shows the corresponding assignment for the derivative ATSAMV7X(N/Q)

Hardware Controller	Interrupt Functions
MCAN_0, BaseAddress: 0x40030000	Can_30_McanIsr_0
MCAN_1, BaseAddress: 0x40034000	Can_30_McanIsr_1

Table 6-2 Hardware Controller – Interrupt Functions

The following table shows the corresponding assignment for the derivative STA1385 (Cut1.0) and STA1385 (Cut2.1)

Hardware Controller	Interrupt Functions
MCAN_0, BaseAddress: 0x50183000	Can_30_McanIsr_0 (using Cut2.1) Can_30_McanIsr_MCAN (using Cut 1.0)
MCAN_1, BaseAddress: 0x50183400	Can_30_McanIsr_1 (using Cut2.1) Can_30_McanIsr_MCAN (using Cut 1.0)

Table 6-3 Hardware Controller – Interrupt Functions

Table 6-4 Shows the corresponding assignment for TDA4VM derivatives.

Node name used in datasheet	Name used in Cfg5	Interrupt Source used in datasheet	ISR name used in Can driver
MCU_MCAN0	M_CAN0	MCU_MCAN0_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_0
MCU_MCAN1	M_CAN1	MCU_MCAN1_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_1
MCAN0	M_CAN2	MCAN0_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_2
MCAN1	M_CAN3	MCAN1_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_3
MCAN2	M_CAN4	MCAN2_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_4
MCAN3	M_CAN5	MCAN3_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_5
MCAN4	M_CAN6	MCAN4_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_6
MCAN5	M_CAN7	MCAN5_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_7
MCAN6	M_CAN8	MCAN6_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_8
MCAN7	M_CAN9	MCAN7_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_9
MCAN8	M_CAN10	MCAN8_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_10
MCAN9	M_CAN11	MCAN9_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_11
MCAN10	M_CAN12	MCAN10_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_12
MCAN11	M_CAN13	MCAN11_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_13
MCAN12	M_CAN14	MCAN12_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_14
MCAN13	M_CAN15	MCAN13_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_15
MCAN14	M_CAN16	MCAN14_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_16
MCAN15	M_CAN17	MCAN15_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_17
MCAN16	M_CAN18	MCAN16_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_18
MCAN17	M_CAN19	MCAN17_MCANSS_MCAN_LVL_INT_0	Can_30_McanIsr_19

Table 6-4 Hardware Controller – Interrupt Functions

Table 6-5 shows the corresponding assignment for the Traveo2 derivatives.

Please not that the consolidated interrupts for this derivative are not supported. Further, Interrupt category 1 is not supported for Traveo2 with IAR compiler. If interrupt category 1 is configured the driver will implement the ISR as a category 0 (void) interrupt.

Node name used in datasheet	Name used in Cfg5 (4/6/8/10ch)	Interrupt Source used in datasheet	ISR name used in Can driver (4/6/8/10ch)
CANFD0_CH0	M_CAN0	canfd_0_interrupts0_0_IRQn	Can_30_McanIsr_0

CANFD0_CH1	M_CAN1	canfd_0_interrupts0_1_IRQn	Can_30_McanIsr_1
CANFD0_CH2	M_CAN(x/2/2/2)	canfd_0_interrupts0_2_IRQn	Can_30_McanIsr_(x/2/2/2)
CANFD0_CH3	M_CAN(x/x/3/3)	canfd_0_interrupts0_3_IRQn	Can_30_McanIsr_(x/x/3/3)
CANFD0_CH4	M_CAN(x/x/x/4)	canfd_0_interrupts0_4_IRQn	Can_30_McanIsr_(x/x/x/4)
CANFD1_CH0	M_CAN(2/3/4/5)	canfd_1_interrupts0_0_IRQn	Can_30_McanIsr_(2/3/4/5)
CANFD1_CH1	M_CAN(3/4/5/6)	canfd_1_interrupts0_1_IRQn	Can_30_McanIsr_(3/4/5/6)
CANFD1_CH2	M_CAN(x/5/6/7)	canfd_1_interrupts0_2_IRQn	Can_30_McanIsr_(x/5/6/7)
CANFD1_CH3	M_CAN(x/x/7/8)	canfd_1_interrupts0_3_IRQn	Can_30_McanIsr_(x/x/7/8)
CANFD1_CH4	M_CAN(x/x/x/9)	canfd_1_interrupts0_4_IRQn	Can_30_McanIsr_(x/x/x/9)

Table 6-5 Hardware Controller – Interrupt Functions

6.2 Interrupt Control

This section describes Interrupt Control.

6.2.1 Interrupt Control by Driver

Parameter 'CAN_30_MCAN_INTLOCK' == CAN_30_MCAN_DRIVER or

CAN_30_MCAN_BOTH)

Parameter 'CAN_30_MCAN_USE_OS_INTERRUPT_CONTROL'== STD_OFF

With this configuration, the interrupt control registers are directly accessed by the driver. In case of the Tricore platform, the SRN registers for the CAN interrupts are accessed. Be aware that this requires access rights (that may be limited by restricted operation modes, memory protection or similar) and also exclusive write access by the driver (the registers must not be written by other modules after the call of Can_30_Mcan_Init()).

6.2.2 Interrupt Control by OS

This feature is only implemented for the Tricore 2G, AWR/TDA3x and AS1 platforms, not implemented on other platforms.

Parameter 'CAN_30_MCAN_INTLOCK' == CAN_30_MCAN_DRIVER or CAN_30_MCAN_BOTH

Parameter 'CAN_30_MCAN_USE_OS_INTERRUPT_CONTROL'== STD_ON

The driver implementation is used to modify all registers as required, but the actual accesses to the interrupt controller registers are performed by services provided by the operating system. This variant also requires that these registers are not written by other software modules after the call of Can_30_Mcan_Init().

Please note
This configuration variant is only supported for MICROSAR4. The used services are provided by the MICROSAR Classic Gen7 OS. If another operating system is used, the calls have to be mapped to corresponding functionalities during integration.

6.2.3 Interrupt Control by Application

Parameter 'CAN_30_MCAN_INTLOCK' == CAN_30_MCAN_APPL or CAN_30_MCAN_BOTH)

Parameter 'CAN_30_MCAN_USE_OS_INTERRUPT_CONTROL' is don't care

If an exclusive write access to the interrupt control registers is not possible, the driver can be configured for an external interrupt handling by means of callout functions. In this case the application must ensure proper disabling and restoring of the used CAN interrupt sources and handling of the wakeup interrupt.

6.3 Extra Registers

Some micro-controllers have extra registers other than the standard MCAN registers. These registers are initialized by calling Can_30_Mcan_Init() API during the initialization phase.

6.4 MCAN Errata

The following Errata (please see ch. 6.2 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	Needless activation of interrupt IR.MRAF. Only activated if “`CAN_BOSCH_ERRATUM_011`“ `is defined as STD_ON.`	2.9.5, 2.9.6, 3.0.0, 3.0.1, 3.1.0
12	Return of receiver from Bus Integration state after Protocol Exception Event. Only activated if “`CAN_BOSCH_ERRATUM_012`“ `is defined as STD_ON.`	2.9.6, 3.0.0, 3.0.1, 3.1.0
13	Message RAM / RAM Arbiter not responding in time. 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. In this case, the respective Rx Buffer or Rx FIFO element holds inconsistent data. **!** 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. The application must assure that the clocking of Host and CAN is appropriate. The CAN Driver does not care about these configuration aspects.	2.9.6, 3.0.0, 3.0.1, 3.1.0, 3.2.0

14	Data loss (payload) in case storage of a received frame has not completed until end of EOF field is reached. The time needed for acceptance filtering and storage of a received message depends on the - 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. 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. Interrupt flag IR.MRAF is not set. 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.	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	Frame transmission in DAR mode. Not considered by the CAN Driver, frame transmission in DAR mode is not supported.	2.9.5, 2.9.6, 3.0.0, 3.0.1
15	Edge filtering causes miss-synchronization when falling edge at Rx input pin coincides with end of integration phase. Not considered by the CAN Driver, Edge Filtering is not supported.	3.1.0, 3.2.0, 3.2.1
16	Configuration of NBTP.NTSEG2 = ‘0’ is not allowed. This erratum is in the responsibility of the application during configuration time and is not considered by the CAN Driver during compile or runtime.	
17	Retransmission in DAR mode due to lost arbitration at the first two identifier bits. Not considered by the CAN Driver, DAR Mode is not supported.	2.9.6, 3.0.0, 3.0.1,

18	Tx FIFO message sequence inversion. Only when Tx Buffers and Tx Hw Fifo is used simultaneously. This erratum is in the responsibility of the application during configuration time and is not considered by the CAN Driver. The workarounds described in the Mcan errata sheet can be applied during configuration (e.g. Only use Tx Hw Fifo, use Can If Buffers for FIFO-like handling instead of Fifo, or avoid transmitting higher priority messages on the FullCan when message sequence inversion should be avoided)	3.1.0, 3.2.0, 3.2.1
19	Unexpected High Priority Message (HPM) interrupt. Not considered by the CAN Driver, High Priority Message interrupt is not supported.	2.9.6, 3.0.0, 3.0.1, 3.1.0, 3.2.0, 3.2.1
20	Message transmitted with wrong arbitration and control fields. Vector do not support any of the suggested workarounds from the errata sheet. Please contact Bosch if more information is needed regarding this.	2.9.6, 3.0.0, 3.0.1, 3.1.0, 3.2.0, 3.2.1, 3.2.2
21	Debug message handling state machine not reset to Idle state when CCCR.INIT is set. Not considered by the driver, Debug on CAN feature is not supported.	2.9.6, 3.0.0, 3.0.1, 3.1.0, 3.2.0, 3.2.1, 3.2.2, 3.2.3, 3.3.0,
22	Message order inversion when transmitting from dedicated Tx Buffers configured with same Message ID. The Vector CAN drivers prioritize the mailboxes according to message ID and not in order of the object ID. Therefore, the erratum #22 does not affect the driver.	2.9.6, 3.0.0, 3.0.1, 3.1.0, 3.2.0, 3.2.1, 3.2.2, 3.2.3,

Table 6-6 MCAN Errata

6.5 Platform Errata

The following Errata must be considered depending on the underlying hardware platform:

6.5.1 Atmel

Compare to the Document Atmel SAM C21E/G/J Datasheet,

Atmel-42365K-SAM-C21_Datasheet_Complete-11/2016, see page 1170, topic 7:

Erratum
7 – The CAN-FD frame format implements Bosch CAN FD Specification V1.0 and is not compatible with ISO11898-1. The CCR.NISO bit has no effect. **Errata reference: 13757** Fix/Workaround: Connect only to CAN-FD networks that support Bosch CAN FD Specification V1.0

Table 6-7 Atmel Erratum Bosch CAN FD

7 API Description

7.1 Interrupt Service Routines provided by CAN

7.1.1 Type of Interrupt Function

- Category 2 (only for OSEK OS or AUTOSAR OS):
  A macro “ISR(Can_30_McanIsr_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_30_MCAN_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_30_MCAN_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_30_MCAN_ENABLE_ISRVOID

7.1.2 CAN ISR API

Prototype	
`void Can_30_McanIsr_<x>(void);`	
Parameter	
---	---
Return code	
---	---
Functional Description	
Handles interrupts of hardware channel <x> for Rx, Tx, BusOff events.	
Particularities and Limitations	
> 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 Can_30_McanIsr_x).	

Table 7-1 MCAN Can_30_McanIsr_<x>

7.2 Services provided by CAN

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

7.2.1 Can_30_Mcan_InitMemory

Prototype	
`void **Can_30_Mcan_InitMemory** (void)`	
Parameter	
-	
Return code	
void	-
Functional Description	
Service initializes module global variables, which cannot be initialized in the startup code. Use this to re-run the system without performing a new start from power on. (E.g.: used to support an ongoing debug session without a complete re-initialization.) Must be followed by a call to "Can_30_Mcan_Init()".	
Particularities and Limitations	
Called by Application.	
	Caution None AUTOSAR API
Call context	
> Should be called while power on initialization before "Can_30_Mcan_Init()" on task level. > This function is Synchronous > This function is Non-Reentrant > Availability: Always	

Table 7-2 Can_30_Mcan_InitMemory

7.2.2 Can_30_Mcan_Init

Prototype	
`void Can_30_Mcan_Init (Can_30_Mcan_ConfigPtrType ConfigPtr)`	
Parameter	
ConfigPtr [in]	Pointer to the configuration data structure. When using the "Multiple ECU" configuration feature, then for each Identity the appropriate "CanConfig_<Identity>"-structure exists and has to be chosen here.
Return code	
void	-
Functional Description	
This function initializes global CAN Driver variables during ECU start-up.	
Particularities and Limitations	
Called by Can Interface. Parameter "ConfigPtr" will be taken into account only for "Multiple ECU Configrutaion" and in Post-Build variant. Disabled Interrupts.	
Call context	
> Has to be called during start-up before CAN communication. > Must be called before calling "Can_30_Mcan_InitController()" but after call of "Can_30_Mcan_InitMemory()". > This function is Synchronous > This function is Non-Reentrant > Availability: Always	

Table 7-3 Can_30_Mcan_Init

7.2.3 Can_30_Mcan_InitController

Prototype	
`void Can_30_Mcan_InitController (uint8 Controller, Can_30_Mcan_ControllerBaudrateConfigPtrType Config)`	
Parameter	
Controller [in]	Number of controller
Config [in]	Pointer to baud rate configuration structure
Return code	
void	-
Functional Description	
Initialization of controller specific CAN hardware. The CAN Driver registers and variables are initialized. The CAN controller is fully initialized and left back within the state "Stop Mode", ready to change to "Running Mode".	
Particularities and Limitations	
Called by CanInterface. Disabled Interrupts.	
Call context	
> Has to be called during the startup sequence before CAN communication takes place but after calling "Can_30_Mcan_Init()". > Must not be called while in "Sleep Mode". > This function is Synchronous > This function is Non-Reentrant > Availability: MICROSAR401 only	

Table 7-4 Can_30_Mcan_InitController

7.2.4 Can_30_Mcan_ChangeBaudrate

Prototype	
`Std_ReturnType` **`Can_30_Mcan_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	> 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	
> Has to be called during the startup sequence before CAN communication takes place but after calling "Can_30_Mcan_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_30_Mcan_ChangeBaudrate

7.2.5 Can_30_Mcan_CheckBaudrate

Prototype	
`Std_ReturnType` **`Can_30_Mcan_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	> E_NOT_OK Baud rate is not available > E_OK Baud rate is available
Functional Description	
This service shall check if the given baud rate is supported of the CAN controller.	
Particularities and Limitations	
Called by Application. The CAN controller must be initialized.	
Call context	
> 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_30_MCAN_CHANGE_BAUDRATE_API == STD_ON")	

Table 7-6 Can_30_Mcan_CheckBaudrate

7.2.6 Can_30_Mcan_SetBaudrate

Prototype	
`Std_ReturnType` **`Can_30_Mcan_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	> 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. (Similar to "Can_30_Mcan_ChangeBaudrate()" but used when identical baud rates are used for different CAN FD settings).	
Particularities and Limitations	
Called by Application.	
Call context	
> 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_30_MCAN_SET_BAUDRATE_API == STD_ON")	

Table 7-7 Can_30_Mcan_SetBaudrate

_InitStruct

7.2.7 Can_30_Mcan_GetVersionInfo

Prototype	
`void` **`Can_30_Mcan_GetVersionInfo`** `(Can_30_Mcan_VersionInfoPtrType VersionInfo)`	
Parameter	
VersionInfo [out]	Pointer to where to store the version information of the CAN Driver. typedef struct { uint16 vendorID; uint16 moduleID; } Std_VersionInfoType;
Return code	
void	-
Functional Description	
Get the version information of the CAN Driver.	
Particularities and Limitations	
Called by Application.	
Call context	
> Only available if "CanVersionInfoApi" is activated. > This function is Synchronous > This function is Reentrant > Availability: "CanVersionInfoApi" is activated ("CAN_30_MCAN_VERSION_INFO_API == STD_ON")	

Table 7-8 Can_30_Mcan_GetVersionInfo

7.2.8 Can_30_McanGetStatus

Prototype	
`uint8 Can_30_Mcan_GetStatus (uint8 Controller)`	
Parameter	
Controller [in]	Number of the controller requested for status information
Return code	
uint8	> CAN_30_MCAN_STATUS_START > CAN_30_MCAN_STATUS_STOP (Bit coded status information) > CAN_30_MCAN_STATUS_INIT > CAN_30_MCAN_STATUS_INCONSISTENT, > CAN_30_MCAN_DEACTIVATE_CONTROLLER (only with "CanRamCheck" active) > CAN_30_MCAN_STATUS_WARNING > CAN_30_MCAN_STATUS_PASSIVE > CAN_30_MCAN_STATUS_BUSOFF > CAN_30_MCAN_STATUS_SLEEP
Functional Description	
Delivers the status of the hardware. Only one of the status bits CAN_30_MCAN_STATUS_SLEEP/ STOP/ START/ BUSOFF/ PASSIVE/ WARNING is set. The CAN_30_MCAN_STATUS_INIT bit is always set if a controller is initialized. CAN_30_MCAN_STATUS_SLEEP has the highest and CAN_30_MCAN_STATUS_WARNING the lowest priority. CAN_30_MCAN_STATUS_INCONSISTENT will be set if one Common CAN channel. is not "Stop" or "Sleep". CAN_30_MCAN_DEACTIVATE_CONTROLLER is set in case the "CanRamCheck" detected an Issue. "status" can be analyzed using the provided API macros: CAN_30_MCAN_HW_IS_OK(status): return "true" in case no warning, passive or bus off occurred. CAN_30_MCAN_HW_IS_WARNING(status): return "true" in case of waning status. CAN_30_MCAN_HW_IS_PASSIVE(status): return "true" in case of passive status. CAN_30_MCAN_HW_IS_BUSOFF(status): return "true" in case of bus off status (may be already false in Notification). CAN_30_MCAN_HW_IS_WAKEUP(status): return "true" in case of not in Sleep Mode. CAN_30_MCAN_HW_IS_SLEEP(status): return "true" in case of Sleep Mode. CAN_30_MCAN_HW_IS_STOP(status): return "true" in case of Stop Mode. CAN_30_MCAN_HW_IS_START(status): return "true" in case of not in Stop Mode. CAN_30_MCAN_HW_IS_INCONSISTENT(status): return "true" in case of an inconsistency between two common CAN channels.	
Particularities and Limitations	
Called by network management or Application.	

	Caution None AUTOSAR API
Call context	
> This function is Synchronous > This function is Non-Reentrant > Availability: ""Can_30_McanGetStatus" is activated ("CAN_30_MCAN_GET_STATUS == STD_ON")	

Table 7-9 Can_30_McanGetStatus

7.2.9 Can_30_Mcan_GetControllerErrorState

Prototype	
`Std_ReturnType` **`Can_30_Mcan_GetControllerErrorState`** `(uint8 Controller, Can_30_Mcan_ErrorStatePtrType ErrorStatePtr)`	
Parameter	
Controller [in]	Number of the controller requested for status information
ControllerModePtr [out]	Pointer to variable to store CAN controller's error state. Must not be NULL.
Return code	
Std_ReturnType	E_NOT_OK Controller mode is not available
Std_ReturnType	E_OK Controller mode is available
Functional Description	
Gets the error state of the given controller.	
Particularities and Limitations	
Configuration Variant(s): CAN_30_MCAN_GET_STATUS == STD_ON (CREQ-178459)	
Call context	
> ANY > This function is Synchronous > This function is Reentrant > Availability: "Can_30_McanGetStatus" is activated ("CAN_30_MCAN_GET_STATUS == STD_ON")	

Table 7-10 Can_30_Mcan_GetControllerErrorState

7.2.10 Can_30_Mcan_GetControllerTxErrorCounter

Prototype	
`Std_ReturnType` **`Can_30_Mcan_GetControllerErrorState`** `(uint8 Controller, Can_30_Mcan_ErrorStatePtrType TxErrorStatePtr)`	
Parameter	
Controller [in]	Number of the controller requested for status information
ControllerModePtr [out]	Pointer to variable to store CAN controller's error state. Must not be NULL.
Return code	
Std_ReturnType	E_NOT_OK Controller mode is not available
Std_ReturnType	E_OK Controller mode is available
Functional Description	
Gets the TX error counter of the given controller.	
Particularities and Limitations	
Configuration Variant(s): CAN_30_MCAN_GET_STATUS == STD_ON (CREQ-178459)	
Call context	
> ANY > This function is Synchronous > This function is Reentrant > Availability: "Can_30_McanGetStatus" is activated ("CAN_30_MCAN_GET_STATUS == STD_ON")	

Table 7-11 Can_30_Mcan_GetControllerTxErrorCounter

7.2.11 Can_30_Mcan_GetControllerRxErrorCounter

Prototype	
`Std_ReturnType` **`Can_30_Mcan_GetControllerErrorState`** `(uint8 Controller, Can_30_Mcan_ErrorStatePtrType RxErrorStatePtr)`	
Parameter	
Controller [in]	Number of the controller requested for status information
ControllerModePtr [out]	Pointer to variable to store CAN controller's error state. Must not be NULL.
Return code	
Std_ReturnType	E_NOT_OK Controller mode is not available
Std_ReturnType	E_OK Controller mode is available
Functional Description	
Gets the RX error counter of the given controller.	
Particularities and Limitations	
Configuration Variant(s): CAN_30_MCAN_GET_STATUS == STD_ON (CREQ-178459)	
Call context	
> ANY > This function is Synchronous > This function is Reentrant > Availability: "Can_30_McanGetStatus" is activated ("CAN_30_MCAN_GET_STATUS == STD_ON")	

Table 7-12 Can_30_Mcan_GetControllerTxErrorCounter

7.2.12 Can_30_Mcan_GetControllerMode

Prototype	
`Std_ReturnType` **`Can_30_Mcan_GetControllerMode`** `(uint8 Controller, Can_ControllerStatePtrType ControllerModePtr)`	
Parameter	
Controller [in]	Number of the controller requested for status information
ControllerModePtr [out]	Pointer to variable to store CAN controller's mode. Must not be NULL.
Return code	
Std_ReturnType	E_NOT_OK Controller mode is not available
Std_ReturnType	E_OK Controller mode is available
Functional Description	
Gets the mode of the given controller.	
Particularities and Limitations	
Configuration Variant(s): CAN_30_MCAN_GET_STATUS == STD_ON (CREQ-178460)	
Call context	
> ANY > This function is Synchronous > This function is Reentrant > Availability: "CanGetStatus" is activated ("CAN_30_MCAN_GET_STATUS == STD_ON")	

Table 7-13 Can_30_Mcan_GetControllerMode

7.2.13 Can_30_Mcan_SetControllerMode

Prototype	
`Can_ReturnType` **`Can_30_Mcan_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	> 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	
> Must not be called within CAN Driver context like RX, TX or Bus Off callouts. > This function is Non-Reentrant > Availability: Always	

Table 7-14 Can_30_Mcan_SetControllerMode

7.2.14 Can_30_Mcan_ResetBusOffStart

Prototype	
`void` **`Can_30_Mcan_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	
> Called while BusOff event handling (Polling or Interrupt context). > This function is Synchronous > This function is Non-Reentrant > Availability: Always	

Table 7-15 Can_30_Mcan_ResetBusOffStart

7.2.15 Can_30_Mcan_ResetBusOffEnd

Prototype	
`void Can_30_Mcan_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	
> Called inside "Can_30_Mcan_SetControllerMode()" while Start transition. > This function is Synchronous > This function is Non-Reentrant > Availability: Always	

Table 7-16 Can_30_Mcan_ResetBusOffEnd

7.2.16 Can_30_Mcan_Write

Prototype	
`Can_ReturnType Can_30_Mcan_Write (Can_HwHandleType Hth, Can_30_Mcan_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	> CAN_NOT_OK transmit unsuccessful > CAN_OK transmit successful > CAN_BUSY transmit could not be accomplished due to the controller is busy.
Functional Description	
Send a CAN message over CAN.	
Particularities and Limitations	
Called by CanInterface. CAN Interrupt locked.	
Call context	
> Called by the CanInterface with at least disabled CAN interrupts. > (Due to data security reasons the CanInterface has to accomplish this and thus it is not needed a further more in the CAN Driver.) > This function is Synchronous > This function is Non-Reentrant > Availability: Always	

Table 7-17 Can_30_Mcan_Write

7.2.17 Can_30_Mcan_CancelTx

Prototype	
`void Can_30_Mcan_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	
> Called by CanTp or Application. > This function is Synchronous	

> This function is Non-Reentrant > Availability: Always

Table 7-18 Can_30_Mcan_CancelTx

7.2.18 Can_30_Mcan_SetMirrorMode

Prototype	
`void **Can_30_Mcan_SetMirrorMode** (uint8 Controller, CddMirror_MirrorModeType mirrorMode)`	
Parameter	
Controller [in]	CAN controller
mirrorMode [in]	Activate or deactivate the mirror mode.
Return code	
void	none
Functional Description	
Activate mirror mode. Switch the Appl_30_Mcan_GenericPreCopy/Confirmation function ON or OFF.	
Particularities and Limitations	
Configuration Variant(s): C_30_MCAN_ENABLE_MIRROR_MODE (user configuration file) Called by "Mirror Mode" CDD. None AUTOSAR API	
Call context	
> ANY > This function is Synchronous > This function is Non-Reentrant	

Table 7-19 Can_30_Mcan_SetMirrorMode

7.2.19 Can_30_Mcan_SetSilentMode

Prototype	
`Std_ReturnType **Can_30_Mcan_SetSilentMode** (uint8 Controller, Can_30_Mcan_SilentModeType silentMode)`	
Parameter	
Controller [in]	CAN controller
silentMode [in]	Activate or deactivate the silent mode with CAN_SILENT_ACTIVE, CAN_SILENT_INACTIVE (Enumeration)
Return code	
Std_ReturnType	E_OK mode change successful.
Std_ReturnType	E_NOT_OK mode change failed.

Functional Description
Activate or deactivate the Silent Mode. Switch to Silent Mode, as a listen only mode without ACK, and deactivate this mode again for regular communication.
Particularities and Limitations
The CAN controller must be in Stop Mode. Configuration Variant(s): CAN_30_MCAN_SILENT_MODE == STD_ON
Call context
> TASK > This function is Synchronous > This function is Non-Reentrant

Table 7-20 Can_30_Mcan_SetSilentMode

7.2.20 Can_30_Mcan_CheckWakeup

Prototype	
`Std_ReturnType` **`Can_30_Mcan_CheckWakeup`** `(uint8 Controller)`	
Parameter	
Controller [in]	Number of the controller to be checked for Wake Up events.
Return code	
Std_ReturnType	> E_OK the given controller caused a Wake Up before. > E_NOT_OK the given controller caused no Wake Up before.
Functional Description	
Service function to check the occurrence of Wake Up events for the given controller (used as Wake Up callback for higher layers).	
Particularities and Limitations	
Called by CanInterface.	
Call context	
> Called while Wakeup validation phase. > This function is Synchronous > This function is Non-Reentrant > Availability: In AR4.x named "Can_30_Mcan_CheckWakeup", in AR3.x named "Can_Cbk_CheckWakeup" (Name mapped by define)	

Table 7-21 Can_30_Mcan_CheckWakeup

7.2.21 Can_30_Mcan_DisableControllerInterrupts

Prototype
`void` **`Can_30_Mcan_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	
> Called within Critical Area handling or out of Application code. > This function is Synchronous > This function is Non-Reentrant > Availability: Always	

Table 7-22 Can_30_Mcan_DisableControllerInterrupts

7.2.22 Can_30_Mcan_EnableControllerInterrupts

Prototype	
`void Can_30_Mcan_EnableControllerInterrupts (uint8 Controller)`	
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	
> Called within Critical Area handling or out of Application code. > This function is Synchronous > This function is Non-Reentrant > Availability: Always	

Table 7-23 Can_30_Mcan_EnableControllerInterrupts

7.2.23 Can_30_Mcan_MainFunction_Write

Prototype	
`void` **`Can_30_Mcan_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_30_Mcan_Write()".	
Call context	
> Called within cyclic TX task. > This function is Synchronous > This function is Non-Reentrant > Availability: Always	

Table 7-24 Can_30_Mcan_MainFunction_Write

7.2.24 Can_30_Mcan_MainFunction_Read

Prototype	
`void` **`Can_30_Mcan_MainFunction_Read`** `(void)`	
Parameter	
-	
Return code	
void	-
Functional Description	
Service function to poll RX events for all controllers and all RX mailboxes to accomplish the RX indication handling (like CanInterface notification). Also used for a delayed read (from task level) of the RX Queue messages which were queued from interrupt context.	
Particularities and Limitations	
Called by SchM.	
Call context	
> Called within cyclic RX task. > This function is Synchronous	

> This function is Non-Reentrant > Availability: Always

Table 7-25 Can_30_Mcan_MainFunction_Read

7.2.25 Can_30_Mcan_MainFunction_BusOff

Prototype	
`void **Can_30_Mcan_MainFunction_BusOff** (void)`	
Parameter	
-	
Return code	
void	-
Functional Description	
Polling of Bus Off events to accomplish the Bus Off handling. Service function to poll Bus Off events for all controllers to accomplish the Bus Off handling (like calling of "CanIf_ControllerBusOff()" in case of Bus Off occurrence).	
Particularities and Limitations	
Called by SchM.	
Call context	
> Called within cyclic BusOff task. > This function is Synchronous > This function is Non-Reentrant > Availability: Always	

Table 7-26 Can_30_Mcan_MainFunction_BusOff

7.2.26 Can_30_Mcan_MainFunction_Wakeup

Prototype	
`void **Can_30_Mcan_MainFunction_Wakeup** (void)`	
Parameter	
-	
Return code	
void	-
Functional Description	
Service function to poll Wake Up events for all controllers to accomplish the Wake Up handling (like calling of "CanIf_SetWakeupEvent()" in case of Wake Up occurrence).	
Particularities and Limitations	
Called by SchM.	

Call context
> Called within cyclic Wakeup task. > This function is Synchronous > This function is Non-Reentrant > Availability: Always

Table 7-27 Can_30_Mcan_MainFunction_Wakeup

7.2.27 Can_30_Mcan_MainFunction_Mode

Prototype	
`void Can_30_Mcan_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_30_Mcan_SetControllerMode()").	
Particularities and Limitations	
Called by SchM.	
Call context	
> Called within cyclic mode change task. > This function is Synchronous > This function is Non-Reentrant > Availability: MICROSAR4x only	

Table 7-28 Can_30_Mcan_MainFunction_Mode

7.2.28 Can_30_Mcan_RamCheckExecute

Prototype	
`void Can_30_Mcan_RamCheckExecute (uint8 Controller)`	
Parameter	
Controller [in]	CAN controller to be checked.
Return code	
void	none
Functional Description	
Check the MCAN Message RAM.	

Check all controller specific and mailbox specific registers by write patterns and read back. Issue notification will appear in this context.
Particularities and Limitations
Has to be called within STOP Mode. Configuration Variant(s): CAN_30_MCAN_RAM_CHECK == CAN_30_MCAN_EXTENDED CREQ-106641
Call context
> TASK > This function is Synchronous > This function is Non-Reentrant

Table 7-29 Can_30_Mcan_RamCheckExecute

7.2.29 Can_30_Mcan_RamCheckEnableMailbox

Prototype	
void **Can_30_Mcan_RamCheckEnableMailbox** (Can_HwHandleType htrh)	
Parameter	
htrh [in]	CAN mailbox to be reactivated.
Return code	
void	none
Functional Description	
Reactivate a mailbox after RamCheck failed. Mailbox will be reactivated by clearing the deactivation flag (see also [7]).	
Particularities and Limitations	
Has to be called within STOP Mode after RamCheck failed (controller is deactivated). Must be followed by Can_30_Mcan_RamCheckEnableController() to activate mailbox and controller. Configuration Variant(s): CAN_30_MCAN_RAM_CHECK == CAN_30_MCAN_EXTENDED CREQ-106641	
Call context	
> TASK > This function is Synchronous > This function is Non-Reentrant	

Table 7-30 Can_30_Mcan_RamCheckEnableMailbox

7.2.30 Can_30_Mcan_RamCheckEnableController

Prototype	
void **Can_30_Mcan_RamCheckEnableController** (uint8 Controller)	
Parameter	
Controller [in]	CAN controller to be reactivated.

Return code	
void	none
Functional Description	
Reactivate CAN cells after RamCheck failed. CAN cell will be reactivated by execute reinitialization.	
Particularities and Limitations	
Has to be called within STOP Mode after RamCheck failed (controller is deactivated). Configuration Variant(s): CAN_30_MCAN_RAM_CHECK == CAN_30_MCAN_EXTENDED CREQ-106641	
Call context	
> TASK > This function is Synchronous > This function is Non-Reentrant	

Table 7-31 Can_30_Mcan_RamCheckEnableController

7.2.31 Appl_30_Mcan_GenericPrecopy

Prototype	
`Can_ReturnType` **`Appl_30_Mcan_GenericPrecopy`** `(uint8 Controller, Can_IdType ID, uint8 DataLength, Can_30_Mcan_DataPtrType DataPtr)`	
Parameter	
Controller [in]	Controller which received the message
ID [in]	ID of the received message (include IDE,FD). In case of extended or mixed ID systems the highest bit (bit 31) is set to mark an extended ID. FD-bit (bit 30) can be masked out with user define CAN_ID_MASK_IN_GENERIC_CALLOUT.
DataLength [in]	Data length of the received message (read only).
pData [in]	Pointer to the data of the received message.
Return code	
Can_ReturnType	> CAN_OK Higher layer indication will be called afterwards (CanIf_RxIndication()). > CAN_NOT_OK Higher layer indication will not be called afterwards.
Functional Description	
Application callback function which informs about all incoming RX messages including the contained data. It can be used to block notification to upper layer. E.g. to filter incoming messages or route it for special handling.	
Particularities and Limitations	
Called by CAN Driver. "pData" is read only and must not be accessed for further write operations.	

The parameter DataLength refers to the received data length by the CAN controller hardware. Note, that the CAN protocol allows the usage of data length values greater than eight (CAN-FD). Depending on the implementation of this callback it may be necessary to consider this special case (e.g. if the data length is used as index value in a buffer write access).
Caution None AUTOSAR API
Call context
> Called within CAN message reception context (Polling or Interrupt). > This function is Synchronous > This function is Non-Reentrant > Availability: "CanGenericPrecopy" is activated ("CAN_30_MCAN_GENERIC_PRECOPY == STD_ON").

Table 7-32 Appl_30_Mcan_GenericPrecopy

7.2.32 Appl_30_Mcan_GenericConfirmation

Prototype	
`Can_ReturnType` **`Appl_30_Mcan_GenericConfirmation`** `(PduIdType PduId)`	
Parameter	
PduId [in]	Handle of the PDU specifying the message.
Return code	
Can_ReturnType	> CAN_OK Higher layer confirmation will be called afterwards (CanIf_TxConfirmation()). > CAN_NOT_OK Higher layer confirmation will not be called afterwards.
Functional Description	
Application callback function which informs about TX messages being sent to the CAN bus.	
Particularities and Limitations	
Called by CAN Driver. "PduId" is read only and must not be accessed for further write operations.	
Caution None AUTOSAR API	
Call context	
> Called within CAN message transmission finished context (Polling or Interrupt). > This function is Synchronous > This function is Non-Reentrant > Availability: "CanGenericConfirmation" is activated ("CAN_30_MCAN_GENERIC_CONFIRMATION == STD_ON") & "CanIfTransmitBuffer" activated (in CanInterface).	

Table 7-33 Appl_30_Mcan_GenericConfirmation

7.2.33 Appl_30_Mcan_GenericConfirmation

Prototype	
`Can_ReturnType **Appl_30_Mcan_GenericConfirmation** (uint8 Controller, Can_30_Mcan_PduInfoPtrType DataPtr)`	
Parameter	
Controller [in]	CAN controller which send the message.
DataPtr [in]	Pointer to a Can_PduType structure including ID (include IDE,FD), DataLength, PDU and data pointer.
Return code	
Can_ReturnType	CAN_OK Higher layer (CanInterface) confirmation will be called. CAN_NOT_OK No further higher layer (CanInterface) confirmation will be called.
Functional Description	
Application callback function which informs about TX messages being sent to the CAN bus. It can be used to block confirmation or route the information to other layers as well.	
Particularities and Limitations	
Called by CAN Driver. A new transmission within this call out will corrupt the DataPtr context. 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.	
	Caution None AUTOSAR API
Call context	
> Called within CAN message transmission finished context (Polling or Interrupt). > This function is Synchronous > This function is Non-Reentrant > Availability: "CanGenericConfirmation" is set to API2 ("CAN_30_MCAN_GENERIC_CONFIRMATION == CAN_30_MCAN_API2").	

Table 7-34 Appl_30_Mcan_GenericConfirmation

7.2.34 Appl_30_Mcan_GenericPreTransmit

Prototype	
`void **Appl_30_Mcan_GenericPreTransmit** (uint8 Controller, Can_30_Mcan_PduInfoPtrType_var DataPtr)`	
Parameter	
Controller [in]	CAN controller on which the message will be send.
DataPtr [in]	Pointer to a Can_PduType structure including ID (include IDE,FD), 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.	
	Caution None AUTOSAR API
Call context	
> Called within "Can_30_Mcan_Write()". > This function is Synchronous > This function is Non-Reentrant > Availability: "CanGenericPretransmit" is activated ("CAN_30_MCAN_GENERIC_PRETRANSMIT == STD_ON").	

Table 7-35 Appl_30_Mcan_GenericPreTransmit

7.2.35 Appl_30_McanCanTimerStart

Prototype	
`void` **`Appl_30_McanCanTimerStart`** `(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 start an observation timer (see chapter Hardware Loop Check / Timeout Monitoring).	
Particularities and Limitations	
Called by CAN Driver.	
	Caution None AUTOSAR API
Call context	
> For context information please refer to chapter "Hardware Loop Check". > This function is Synchronous > This function is Non-Reentrant > Availability: "CanHardwareCancelByAppl" is activated ("CAN_30_MCAN_HW_LOOP_SUPPORT_API == STD_ON").	

Table 7-36 Appl_30_McanCanTimerStart

7.2.36 Appl_30_McanCanTimerLoop

Prototype	
`Can_ReturnType **Appl_30_McanCanTimerLoop** (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	
Can_ReturnType	> CAN_NOT_OK when loop shall be broken (observation stops) > CAN_NOT_OK should only be used in case of a timeout occurs due to a hardware issue. > After this an appropriate error handling is needed (see chapter Hardware Loop Check / Timeout Monitoring). > CAN_OK when loop shall be continued (observation continues)
Functional Description	
Service function to check (against generated max loop value) whether a hardware loop shall be continued or broken.	
Particularities and Limitations	
Called by CAN Driver.	
	Caution None AUTOSAR API
Call context	
> For context information please refer to chapter "Hardware Loop Check". > This function is Synchronous > This function is Non-Reentrant > Availability: "CanHardwareCancelByAppl" is activated ("CAN_30_MCAN_HW_LOOP_SUPPORT_API == STD_ON").	

Table 7-37 Appl_30_McanCanTimerLoop

7.2.37 Appl_30_McanCanTimerEnd

Prototype	
`void **Appl_30_McanCanTimerEnd** (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	
> For context information please refer to chapter "Hardware Loop Check". > This function is Synchronous > This function is Non-Reentrant > Availability: "CanHardwareCancelByAppl" is activated ("CAN_30_MCAN_HW_LOOP_SUPPORT_API == STD_ON").	

Table 7-38 Appl_30_McanCanTimerEnd

7.2.38 Appl_30_McanCanInterruptDisable

Prototype	
`void Appl_30_McanCanInterruptDisable (uint8 Controller)`	
Parameter	
Controller [in]	Number of the controller for the CAN interrupt lock.
Return code	
void	-
Functional Description	
Service function to support the disabling of CAN Interrupts by the application. E.g.: the CAN Driver itself should not access the common Interrupt Controller due to application specific restrictions (like security level etc.). Or the application like to be informed because of an CAN interrupt lock.	
Particularities and Limitations	
Called by CAN Driver.	
	Caution None AUTOSAR API
Call context	
> Called by the CAN Driver within "Can_30_Mcan_DisableControllerInterrupts()".	

> This function is Synchronous > This function is Non-Reentrant > Availability: "CanInterruptLock" is set to APPL or BOTH ("CAN_30_MCAN_INTLOCK == CAN_30_MCAN_APPL" or "CAN_30_MCAN_INTLOCK == CAN_30_MCAN_BOTH").

Table 7-39 Appl_30_McanCanInterruptDisable

7.2.39 Appl_30_McanCanInterruptRestore

Prototype	
`void Appl_30_McanCanInterruptRestore (uint8 Controller)`	
Parameter	
Controller [in]	Number of the controller for the CAN interrupt unlock.
Return code	
void	-
Functional Description	
Service function to support the enabling of CAN Interrupts by the application. E.g.: the CAN Driver itself should not access the common Interrupt Controller due to application specific restrictions (like security level etc.). Or the application like to be informed because of an CAN interrupt lock.	
Particularities and Limitations	
Called by CAN Driver.	
	Caution None AUTOSAR API
Call context	
> Called by the CAN Driver within "Can_30_Mcan_EnableControllerInterrupts()". > This function is Synchronous > This function is Non-Reentrant > Availability: "CanInterruptLock" is set to APPL or BOTH ("CAN_30_MCAN_INTLOCK == CAN_30_MCAN_APPL" or "CAN_30_MCAN_INTLOCK == CAN_30_MCAN_BOTH").	

Table 7-40 Appl_30_McanCanInterruptRestore

7.2.40 Appl_30_Mcan_CanOverrun

Prototype	
`void Appl_30_Mcan_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	
> 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_30_MCAN_OVERRUN_NOTIFICATION == CAN_30_MCAN_APPL").	

Table 7-41 Appl_30_Mcan_CanOverrun

7.2.41 Appl_30_Mcan_CanFullCanOverrun

Prototype	
`void Appl_30_Mcan_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	
> 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_30_MCAN_OVERRUN_NOTIFICATION == CAN_30_MCAN_APPL").	

Table 7-42 Appl_30_Mcan_CanFullCanOverrun

7.2.42 Appl_30_Mcan_CanCorruptMailbox

Prototype	
`void Appl_30_Mcan_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_30_Mcan_InitController()") about a defect mailbox within the CAN cell.	
Particularities and Limitations	
Called by CAN Driver.	
Caution None AUTOSAR API	
Call context	
> Call within controller initialization. > This function is Synchronous > This function is Non-Reentrant > Availability: "CanRamCheck" set to "MailboxNotifiation" ("CAN_30_MCAN_RAM_CHECK == CAN_30_MCAN_NOTIFY_MAILBOX").	

Table 7-43 Appl_30_Mcan_CanCorruptMailbox

7.2.43 Appl_30_Mcan_CanRamCheckFailed

Prototype	
`uint8 Appl_30_Mcan_CanRamCheckFailed (uint8 Controller)`	
Parameter	
Controller [in]	Number of the controller for which the check failed
Return code	
uint8	> action With this "action" the application can decide how to proceed with the initialization. > CAN_30_MCAN_DEACTIVATE_CONTROLLER - deactivate the controller > CAN_30_MCAN_ACTIVATE_CONTROLLER - activate the controller
Functional Description	
This function will notify the application (during "Can_30_Mcan_InitController()") about a defect CAN controller due to a previous failed mailbox check.	

Particularities and Limitations
Called by CAN Driver.
Caution None AUTOSAR API
Call context
> Call within controller initialization. > This function is Synchronous > This function is Non-Reentrant > Availability: "CanRamCheck" set to "Active" or "MailboxNotifiation" ("CAN_30_MCAN_RAM_CHECK != CAN_30_MCAN_NONE").

Table 7-44 Appl_30_Mcan_CanRamCheckFailed

7.2.44 Appl_30_McanCanInitPostProcessing

Prototype

`void` **`Appl_30_McanCanInitPostProcessing`** `(CAN_30_MCAN_HW_CHANNEL_CANTYPE_ONLY)`

Parameter	
Controller [in]	Number of the controller for which the check failed
Return code	
void	none

Functional Description

Service function to

a) overwrite the previously set initialization values for the bit timing, taken from the generated data, with customer specific values:

For your convenience, the following access macro is supported:

- **CanBtpReg**(controller): the (N)BTP register of the specified CAN channel can be set according to the register definition (see Hardware Manufacturer Document in ch. 2).

Example: CanBtpReg(Controller) = 0x00070F70u;

or CanBtpReg(0) = 0x00070F70u; (when using 'Optimize for one controller').

b) bypass or configure the CAN Calibration Unit (CCCU) where available:

The CCCU is only available when using SPC58xx derivatives. Within this callback you are requested toeither configure or bypass the CCCU. If so, you must assure by your configuration that the first CAN channel being initialized is „MCAN_2“(Base Address 0xF7EE8000).

Example (for SPC58EC80):

```
// CCCR INIT and CCE bits aleady set by CAN Driver during
   initialization
CCCU.CCFG |= 0x00000040ul;    // CCFG.BCC = „1‟ Bypass CCCU
MCAN.CCCR &= 0xFFFFFFFBul;    // CCCR.ASM = „0‟ Reset Restricted Mode
```

Particularities and Limitations

Called by CAN Driver at the end of the CAN Driver initialization.

none
Caution None AUTOSAR API It is the responsibility of the application to assure that the register values are consistent with the release of the underlying derivative.
Call context
> Called within controller initialization. > This function is Synchronous > This function is Non-Reentrant > Availability: Only available if 'C_30_MCAN_ENABLE_INIT_POST_PROCESS' is defined via a user-config file.

Table 7-45 Appl_30_McanCanInitPostProcessing

7.2.45 Appl_30_McanCanEccInit

Prototype	
`uint8 Appl_30_McanCanEccInit (void)`	
Parameter	
void	-
Return code	
> uint8	> kCan_30_McanOk Initialization of the CAN Message RAM was successful > kCan_30_McanFailed Initialization of the CAN Message RAM failed
Functional Description	
Gives a notification to the application at the correct point of time to initialize the Can message RAM during the initialization of the CAN Driver	
Particularities and Limitations	
Called by CAN Driver.	
Caution None AUTOSAR API	
Call context	
> Call within controller initialization. > This function is Synchronous > This function is Non-Reentrant > Availability: "CanEccInit" set to "False"	

Table 7-46 Appl_30_McanCanEccInit

7.2.46 Appl_30_McanCanPowerOnGetBaseAddress()

Prototype		
`uint32 Appl_30_McanCanPowerOnGetBaseAddress (uint32 phyAddr, uint16 size)`		
Parameter		
phyAddr [in]	-	The physical address (controller base address) to be converted into address
Size[in]	-	The size of the memory block mapped to virtual memory
Return code		
> uint32	> Returns the virtual address to be used by the driver	
Functional Description		
Callback function to set the virtual address in the driver during Can_30_Mcan_Init()		
Particularities and Limitations		
Application function which is called by the CAN driver during Can_30_Mcan_Init() to set the corresponding virtual address of a physical address (CAN controller hardware base address, shared message RAM base address, global base address, and subsystem base address) depending on the underlying hardware. Configuration Variant(s): C_30_MCAN_ENABLE_UPDATE_BASE_ADDRESS Cfg5: Can/CanGeneral/CanUseVirtualAddressing None AUTOSAR API		
Caution None AUTOSAR API		
Call context		
> Call within controller initialization. > This function is Synchronous > This function is Non-Reentrant		

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])

Application (optional non AUTOSAR)	**Appl_30_Mcan_GenericPrecopy** **Appl_30_Mcan_GenericConfirmation** **Appl_30_Mcan_GenericPreTransmit** **Appl_30_McanCanTimerStart/Loop/End** **Appl_30_Mcan_CanRamCheckFailed, Appl_30_Mcan_CanCorruptMailbox** **Appl_30_McanCanInterruptDisable/Restore** **Appl_30_Mcan_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 sucessfully changed. **CanIf_RamCheckCorruptMailbox** Notification if RAM check detects a corrupt mailbox. **CanIf_RamCheckCorruptController** Notification if RAM check detects a corrupt CAN channel.
Os (MICROSAR4x)	**OS_TICKS2MS_<counterShortName>()** Os macro to get timebased ticks from counter. **GetElapsedValue** Get elapsed tick count. **GetCounterValue** Get tick count start. **osDisableInterruptSource(ISRType IsrId)** Disable the CAN interrupt in the interrupt controller. **osEnableInterruptSource(ISRType IsrId, boolean ClearPending)** Enable the CAN interrupt in the interrupt controller and optionally clear pending requests. **osIsInterruptSourceEnabled(ISRType IsrId, boolean *IsEnabled)** Get the CAN interrupt enable state. A symbolic name that equals the corresponding ISR name is passed as IsrId. The definitions of ISRType as well as the symbolic names are provided by Os.h.

Table 7-47 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.

8.1 Pre-Compile Parameters

Settings have to be available before compilation:

> Version API (Can_30_Mcan_GetVersionInfo() activation)
```
#define CAN_30_MCAN_VERSION_INFO_API          STD_ON/STD_OFF
```
> DET (development error detection)
```
#define CAN_30_MCAN_DEV_ERROR_DETECT          STD_ON/STD_OFF
```
> Hardware Loop Check (timeout monitoring)
```
#define CAN_30_MCAN_HARDWARE_CANCELLATION     STD_ON/STD_OFF
```
> Polling modes: Tx confirmation, Reception, Wakeup, BusOff
```
#define CAN_30_MCAN_TX_PROCESSING
CAN_30_MCAN_INTERRUPT/CAN_30_MCAN_POLLING
#define CAN_30_MCAN_RX_PROCESSING
CAN_30_MCAN_INTERRUPT/CAN_30_MCAN_POLLING
#define CAN_30_MCAN_BUSOFF_PROCESSING
CAN_30_MCAN_INTERRUPT/CAN_30_MCAN_POLLING
#define CAN_30_MCAN_WAKEUP_PROCESSING
CAN_30_MCAN_INTERRUPT/CAN_30_MCAN_POLLING
#define CAN_30_MCAN_INDIVIDUAL_PROCESSING     STD_ON/STD_OFF
```
> Multiplexed Tx (external PIA – by usage of multiple Tx mailboxes)
```
#define CAN_30_MCAN_MULTIPLEXED_TRANSMISSION  STD_ON/STD_OFF
```
> Tx Hardware FIFO
```
#define CAN_30_MCAN_TX_HW_FIFO                STD_ON/STD_OFF
```
> Configuration Variant (define the configuration type when using post build variant)
```
#define CAN_30_MCAN_ENABLE_SELECTABLE_PB
```
> Use Generic Precopy Function (None AUTOSAR feature)
```
#define CAN_30_MCAN_GENERIC_PRECOPY           STD_ON/STD_OFF
```
> Use Generic Confirmation Function (None AUTOSAR feature)
```
#define CAN_30_MCAN_GENERIC_CONFIRMATION      STD_ON/STD_OFF
```
> Use Rx Queue Function (None AUTOSAR feature)
```
#define CAN_30_MCAN_RX_QUEUE               STD_ON/  (not supported)
                                           STD_OFF
```
> Used ID type (standard/extended or mixed ID format)
```
#define CAN_30_MCAN_EXTENDED_ID              STD_ON/STD_OFF
#define CAN_30_MCAN_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_30_MCAN_RX_FULLCAN_OBJECTS               STD_ON/STD_OFF
#define CAN_30_MCAN_TX_FULLCAN_OBJECTS               STD_ON/STD_OFF
#define CAN_30_MCAN_RX_BASICCAN_OBJECTS              STD_ON/STD_OFF
```

> Use Multiple BasicCAN objects

```
#define CAN_30_MCAN_MULTIPLE_BASICCAN                STD_ON/STD_OFF
```

> Optimizations

```
#define CAN_30_MCAN_ONE_CONTROLLER_OPTIMIZATION  STD_ON/STD_OFF
#define CAN_30_MCAN_DYNAMIC_FULLCAN_ID               STD_ON/STD_OFF
```

> Usage of nested CAN interrupts

```
#define CAN_30_MCAN_NESTED_INTERRUPTS                STD_ON/STD_OFF
```

> Use Multiple ECU configurations

```
#define CAN_30_MCAN_MULTI_ECU_CONFIG                 STD_ON/STD_OFF
```

> Use RAM Check (verify mailbox buffers)

```
#define CAN_30_MCAN_RAM_CHECK        CAN_30_MCAN_NONE/
                                     CAN_30_MCAN_NOTIFY_ISSUE/
                                     CAN_30_MCAN_NOTIFY_MAILBOX
```

> Use Overrun detection

```
#define CAN_30_MCAN_OVERRUN_NOTIFICATION      CAN_30_MCAN_NONE/
                                              CAN_30_MCAN_DET/
                                              CAN_30_MCAN_APPL
```

> Tx Cancellation of Identical IDs

```
#define CAN_30_MCAN_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.

Caution

Since the Generation Tool does not know which MCAN Hardware Revision for the selected derivative you are actually using, this has to be specified in addition.

Please see below the supported values for the different hardware revisions:

MCAN_REV_10: MCAN Release 1, no CAN-FD support, no Rx FullCAN support
MCAN_REV_20: MCAN Release 2, CAN-FD support with higher bitrates
MCAN_REV_30: MCAN Release 3, CAN-FD support with higher bitrates and extended data length
MCAN_REV_310: MCAN Release 3, Step 1, SubStep 0, non-compatibility change to previous revisions
MCAN_REV_315: MCAN Release 3, Step 1, SubStep 5, CAN-FD support with ISO-11898-1 compatibility

The MCAN Revisions 3.2.x are compatible with MCAN_REV_315 and thus not mentioned separately.

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

9.2 Hardware Limitations

9.2.1 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.

9.2.2 Rx side

SREQ00014271 "message reception shall use overwrite mode" is not fulfilled for FullCAN messages due to hardware limitations. Rx BasicCan is only supported in blocking mode.

9.2.3 Used resources

Please note that the theoretical possible maximum configuration for the derivatives often 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 24320 bytes (4864 * 5) can theoretically be configured but less RAM is physically available (e.g.: 16 KByte per CAN channel). You are requested to reduce the areas according to your needs.

Please note that in case of CAN-FD with data lengths greater than 8 data bytes corresponding Message RAM sizes have to be taken into consideration.

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”).

9.2.4 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 and use your own initialization instead of.

To disable the CAN Driver internal initialization, deactivate the feature “CanEccInit” in DaVinci Configurator (Can/CanGeneral/CanEccInit).

When this feature is deactivated, the can driver will issue an application callout, Appl_30_McanCanEccInit at the appropriate point of time during the initialization of the Can driver.

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

9.3 Vector Extensions

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

10 Glossary and Abbreviations

10.1 Glossary

Term	Description
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
CFG5	DaVinci Configurator Pro
DEM	Diagnostic Event Manager
DET	Development Error Tracer
ECU	Electronic Control Unit
HIS	Hersteller Initiative Software
ISR	Interrupt Service Routine
MICROSAR Classic	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 objects)
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