MICROSAR CAN Driver

Technical Reference

Renesas RH850/P1x-C MCAN
Version 3.00.01

Authors	P. Herrmann
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” - 4.10 “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.

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	REL2015 0701
[7]	Appl. Note AN-ISC-8-1190 CAN Self Diag	1.1.0

Table 1-2 Reference Documents

1.3 Scope of the Document

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

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
R7F701325 R7F701327 R7F701328 R7F701329 R7F701370A R7F701370B R7F701371 R7F701372 R7F701372A R7F701373 R7F701373A R7F701374 R7F701374A	GHS	RH850/P1x-C Group RH850/P1x-C Group RH850/P1x-C Group	Rev. 0.60, Sep. 2014 Nov, 2014 Rev. 0.10 Jan, 2016 Rev.1.00

Table 2-1 Supported Hardware Overview

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

Compiler: List of Compilers the CAN Driver is working with

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

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

3 Introduction

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

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

Supported AUTOSAR Release*:	3 and 4	
Supported Configuration Variants:	Pre-Compile, Link-Time, Post-Build Loadable, Post-Build Selectable (MICROSAR Identity Manager)	
Vendor ID:	CAN_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	CAN_MODULE_ID	80 decimal (according to ref. [2])
AR Version:	CAN_AR_RELEASE_MAJOR_VERSION CAN_AR_RELEASE_MINOR_VERSION CAN_AR_RELEASE_REVISION_VERSION	AUTOSAR Release version BCD coded
SW Version:	CAN_SW_MAJOR_VERSION CAN_SW_MINOR_VERSION CAN_SW_PATCH_VERSION	MICROSAR CAN mudule 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.

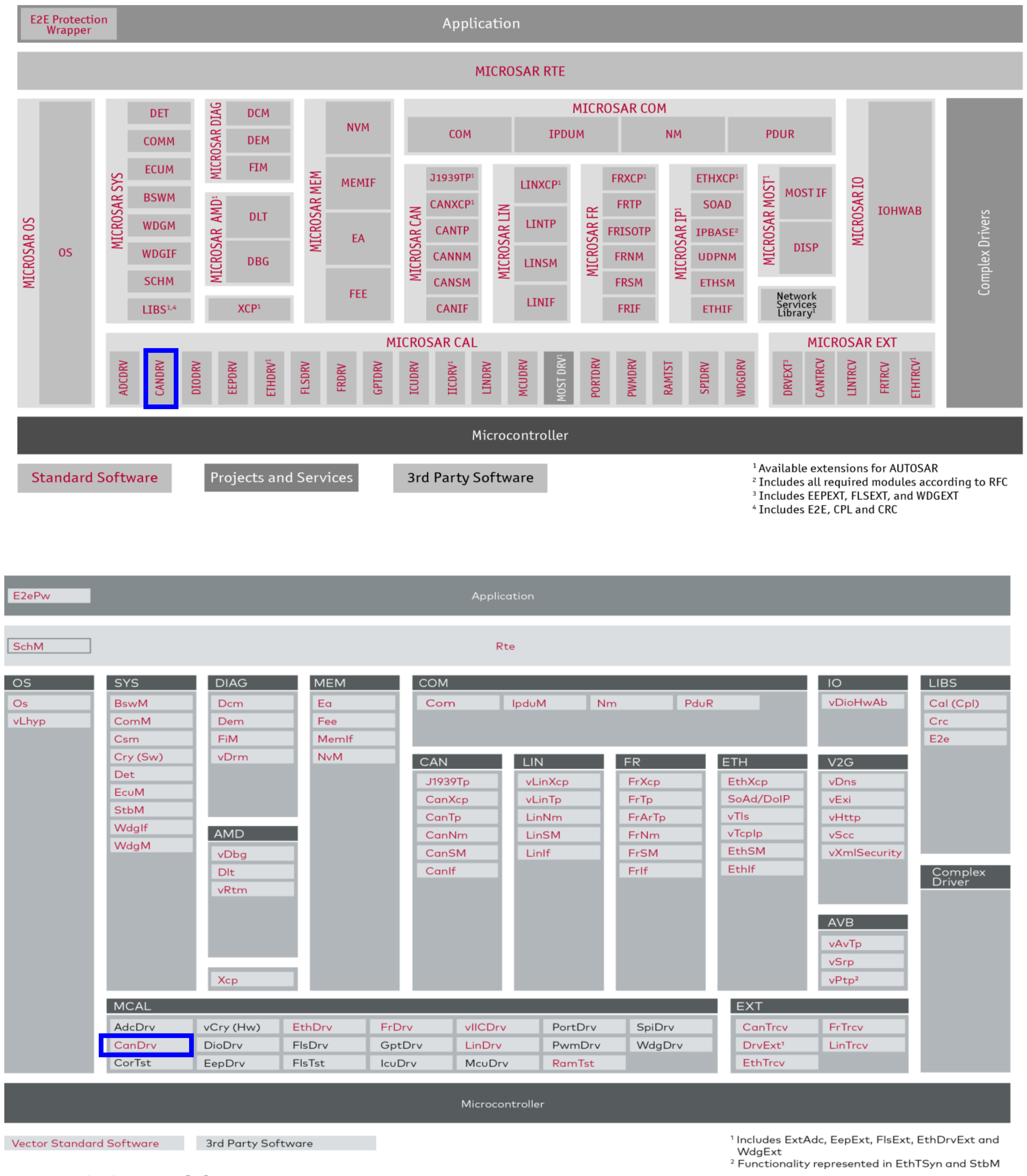


Figure 3-1 AUTOSAR architecture

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

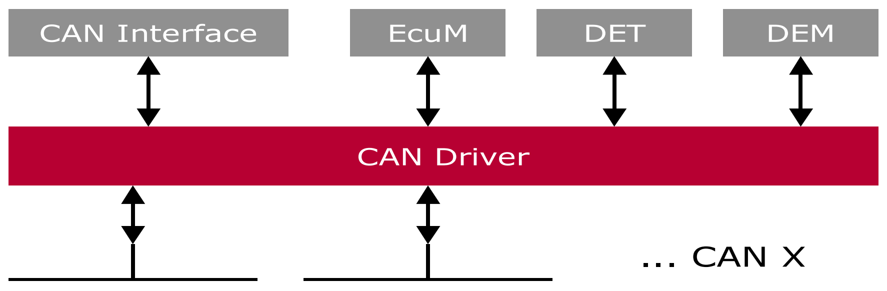


Figure 3-2 Interfaces to adjacent modules of the CAN

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	GENy	CFG5
Initialization			
Driver	General driver initialization function Can_Init()	■	■
Controller	Controller specific initialization function Can_InitController().	■	■
Communication			
Transmission	Transmitting CAN frames.	■	■
Transmit confirmation	Callback for successful Transmission.	■	■
Reception	Receiving CAN frames.	■	■
Receive indication	Callback for receiving Frame.	■	■
Controller Modes			
Sleep Mode	Controller supports Sleep Mode (power saving).	□	□
Wakeup over CAN	Controller supports wakeup over CAN.	□	□
Stop Mode	Controller supports Stop Mode (passive to CAN bus).	■	■
Bus Off detection	Callback for Bus Off event.	■	■
Silent Mode	Support Silent Mode where the controller only listens passive.	□	■
Polling Modes			
Tx confirmation	Support Polling Mode for Transmit confirmation.	■	■
Reception	Support Polling Mode for Reception.	■	■
Wakeup	Support Polling Mode for Wakeup event.	□	□
Bus Off	Support Polling Mode for Bus Off event.	■	■
Mode	MICROSAR4x only: Support Polling Mode for mode transition.	■	■
Mailbox objects			
Tx BasicCAN	Standard mailbox to send CAN frames (Used by CAN Interface data queue).	■	■

Multiplexed Tx	Using 3 mailboxes for Tx BasicCAN mailbox (external priority inversion avoided).	■	■
Tx FullCAN	Separate mailbox for special Tx message used.	■	■
Maximum amount	Available amount of mailboxes.	32	32
Rx FullCAN	Separate mailbox for special Rx message used.	■	■
Maximum amount	Available amount of mailboxes.	64	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)	1 (64) 2(128)
Others			
DEM	Support Diagnostic Event Manager (error notification).	■	■
DET	Support Development Error Detection (error notification).	■	■
Version API	API to read out component version.	■	■
Maximum supported Controllers	Maximum amount of supported Controllers (hardware channels).	8	8
Cancellation of Tx objects	Support of Tx Cancellation (out of hardware). Avoid internal priority inversion.	■	■
Identical ID cancellation	Tx Cancellation also for identical IDs.	■	■
Standard ID types	Standard Identifier supported (Tx and Rx).	■	■
Extended ID types	Extended Identifier supported (Tx and Rx).	■	■

Mixed ID types	Standard and Extended Identifier supported (Tx and Rx).	■	■
CAN FD Mode1	FD frames with baudrate switch supported (Tx and Rx).	■	■
CAN FD Mode2	FD frames up to 64 data bytes supported (Tx and Rx).	□	■****
Hardware Loop Check (Timeout monitoring)	To avoid possible endless loops (occur by hardware issue).	■	■
AutoSar extensions			
Individual Polling	Support individual Polling Mode (selectable for each mailbox separate).	■*	■*
Multiple Rx Basic CAN	Support Multiple BasicCAN objects. This gives the possibility to use additionally Fifo-1 with 64 additional elements. By optimizing the acceptance filtering overruns can be avoided .	■*	■*
Multiple Tx Basic CAN	Support Multiple Tx BasicCAN objects. Used to send different Tx groups over separate mailboxes with different buffering behavior (see Can Interface).	□*	■*
Rx Queue	Support Rx Queue. This offers the possibility to buffer received data in interrupt context but handle it later asynchronous in the polling task.	■*	■*
Secure Rx Buffer used	Special hardware buffer used to save received data temporarily.	□	□
Hardware Loop Check by Application	Hardware Loop Check can be defined to be done by application (special API available)	■	■
Configurable “Nested CAN Interrupts”	Nested CAN interrupts allowed and can be also switched to none-nested.	■	■
Report CAN_E_TIMEOUT DEM as DET	Report CAN_E_TIMEOUT (Hardware Loop Check / Timeout monitoring) to DET instead of DEM.	■	■
Support Mixed ID	Force CAN Driver to handle Mixed ID (standard and extended ID) at pre-compile-time to expand the ID type later on.	■	■

Optimize for one controller	Activate this for 1 controller systems when you never will expand to multi-controller. So that the CAN Driver works more efficient	■	■
Dynamic FullCAN Tx ID (***)	Always write FullCAN Tx ID within Can_Write() API function. Deactivate this to optimize code when you do not use FullCAN Tx objects dynamically.	■	■
Size of Hw HandleType	Support 8-bit or 16-bit Hardware Handles depending on the hardware usage.	■	■
Generic PreCopy	Support a callback function for receiving any CAN message (following callbacks could be suppressed)	■	■
Generic Confirmation	Support a callback function for successful transmission of any CAN message (following callbacks could be suppressed)	■	■
Get Hardware Status	Support a API to get hardware status Information (see Can_GetStatus())	■	■
Interrupt Category selection	Support Category 1 or Category 2 Interrupt Service Routines for OS	■	■
Common CAN	Support merge of 2 controllers in hardware to get more Rx FullCAN objects	□	□
Overrun Notification	Support DET or Application notification caused by overrun (overwrite) of an Rx message. 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.	■	■
RAM check	Support CAN mailbox RAM check	■	■

Extended RAM Check	Support extended RAM check. Handling of individual deactivated mailboxes and controllers.	□	■
Multiple ECU configurations (***)	The feature Multiple ECU is usually used for nodes that exist more than once in a car. At power up the application decides which node should be realized.	■	□
Generic PreTransmit	Support a callback function with pointer to Data, right before this data will be written in Hardware mailbox buffer to send. (Use this to change data or cancel transmission)	■	■

Table 4-1 Supported features

■	Feature is supported
□	Feature is not supported
*	HighEnd Licence only
**	Project specific (may not be available)
***	Not supported or cannot be configured for AutoSar version 4
****	Only available for MicroSar4

4.2 Initialization

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

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

MICROSAR401 only: baud rate settings given by Can_InitController parameter.

Can_InitController() initializes the controller, given as parameter, and can also be used to reinitialize. After this call the controller stays in Stop Mode until the CAN Interface changes to Start Mode.

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

4.3 Communication

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

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

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

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

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

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

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

4.3.1 Mailbox Layout

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

Hardware object number	Hardware object type	Amount of hardware objects	Description
0 … N	**Tx FullCAN**	0 … 31 max. (0 … 29 in case of multiplexed transmission)	These objects are used to transmit specific message IDs. The user must define statically in the generation tool which CAN message IDs are located in Tx FullCAN objects. The generation tool assigns the message IDs to the FullCAN hardware objects.
(N+1) … M	**Tx BasicCAN**	1 or 3 (3 in case of multiplexed transmission)	All other CAN message IDs are transmitted via the Tx Basic object. If the transmit message object is busy, the transmit requests are stored in the CAN Interface queue (if activated).
(M+1) … O	**Unused**	0 … 95	These objects are not used. It depends on the configuration of receive and transmit objects how many unused objects are available.
O…P	**Rx FullCAN**	0 … 64	These objects are used to receive specific CAN messages. The user defines statically (Generation Tool) that a CAN message should be received in a FullCAN message object. The Generation Tool distributes the messages to the FullCAN objects.

96	**Rx BasicCAN**	FIFO-0 with max. 64 entries	All CAN message IDs, depending on the acceptance filter match, are received via the Rx BasicCAN message object through Rx FIFO 0. Each Rx Basic message object consists of 64 message buffers. 128 acceptance filters are available for standard IDs and 64 acceptance filters are available for extended IDs. In case of mixed ID Mode 128+64 = 192 filters are available. Please note that this maximum amount of filters is also used for FIFO-1 if available.
97	**Rx BasicCAN**	FIFO-1 with max. 64 entries	All CAN message IDs, depending on the acceptance filter match, are received via the Rx BasicCAN message objects through Rx FIFO 1. Each Rx Basic message object consists of 64 message buffers. 128 acceptance filters are available for standard IDs and 64 acceptance filters are available for extended IDs. In case of mixed ID Mode 128+64 = 192 filters are available. Please note that this maximum amount of filters is also used for FIFO-0.

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

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

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

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

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

4.3.3 Acceptance Filter for BasicCAN

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

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

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

4.3.4 Remote Frames

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

4.4 States / Modes

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

The following mode changes are supported:

`CAN_T_START`

`CAN_T_STOP`

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

4.4.1 Start Mode (Normal Running Mode)

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

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

4.4.2 Stop Mode

If Stop Mode is requested, either by software or by going BusOff, then the CAN module is switched into INIT mode. In this mode message transfer from and to the CAN bus is stopped, the status of the CAN bus transmit output is recessive (HIGH).

Going to Stop Mode does not change any configuration register.

4.4.3 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 ApplCanClockStop(CanChannel)     ApplCanClockStopAcknowledged(CanChannel)
/* will be called when the application is allowed to turn off the clocks for CAN and Host */
#define ApplCanClockStart(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_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.

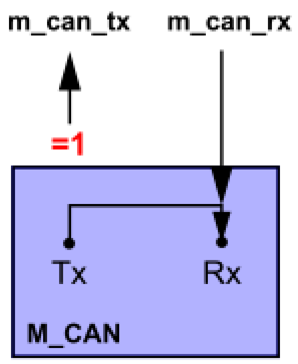


Figure 4-1 Bus Monitoring Mode.

Caution
With activated "Silent Mode" do not use any other API than
Can_SetSilentMode("CAN_SILENT_INACTIVE"),
Can_SetControllerMode("START" or "STOP") or
Can_ChangeBaudrate() or Can_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_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 "`ApplCanInitPostProcessing()`" 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. 7.2.42).

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

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

4.6 CAN Interrupt Locking

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

4.7 Main Functions

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

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

If the Rx Queue feature is activated then the queue is filled in interrupt or polling context, like configured. But the processing (indications) will be done in “`Can_MainFunction_Read()`” context.

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

4.8 Error Handling

4.8.1 Development Error Reporting

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

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

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

Errors reported to DET:

Error ID	Short Description
CAN_E_PARAM_POINTER	API gets an illegal pointer as parameter.
CAN_E_PARAM_HANDLE	API gets an illegal handle as parameter
CAN_E_PARAM_DLC	API gets an illegal DLC as parameter
CAN_E_PARAM_CONTROLLER	API gets an illegal controller as parameter
CAN_E_UNINIT	Driver API is used but not initialized
CAN_E_TRANSITION	Transition for mode change is illegal
CAN_E_DATALOST (value: 0x07, AutoSar extension)	Rx overrun (overwrite) detected
CAN_E_PARAM_BAUDRATE (value: 0x08, AutoSar extension)	Selected Baudrate is not valid
CAN_E_RXQUEUE (value: 0x10, AutoSar extension)	Rx Queue overrun (Last received message is lost and will not be received. Avoid this by increasing the queue size)
CAN_E_TIMEOUT_DET (value: 0x11, AutoSar extension)	Same as CAN_E_TIMEOUT for DEM but this is notified to DET due to switch “CAN_DEV_TIMEOUT_DETECT” is set to STD_ON (see configuration options)
CAN_E_GENDATA (value:0x12, AutoSar extension)	Standardized issue for inconsistent generated data
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.

Table 4-2 Errors reported to DET

API from which the errors are reported to DET:

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

4.8.1.2 Overrun/Overwrite Notification

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

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_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_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 issue. This feature is configurable see Chapter 7 and also 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 internal 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 has to be handled by application like described below.

Loop Name / source	Short Description
kCanLoopInit	This channel dependent loop is called in Can_InitController and is processed as long as the CAN cell does not enter resp. leave the configuration mode. While entering the configuration mode, message transfer from and to the CAN bus is stopped, the status of the CAN bus transmit output is recessive. There is a delay from writing to a command register until the update of the related status register bits due to clock domain crossing (Host and CAN clock). Therefore the programmer has to assure that the previous value written to INIT has been accepted. If the loop cancels, try to reinitialize the controller again 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
kCanLoopSleep	When Clock Stop is requested then all pending transfer requests are completed first. When the CAN bus reached idle then Clock Stop will be acknowledged. (See also „kCanLoopClockStop" below, but only used for Bosch Erratum #7)
kCanLoopClockStop	When Clock Stop is requested then all pending transfer requests are completed first. When the CAN bus reached idle then Clock Stop will be acknowledged. (Please see also ch. 4.4.3 Power Down Mode)
kCanLoopRxFifo	This channel dependent loop is called in CanInterruptRx and is processed until the Rx FIFO becomes empty. The loop is delayed if the controller receives a burst of messages. The maximum expected duration is the time needed until all messages in the reception FIFO are confirmed. If the loop cancels 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)

4.8.3.3 Loops used for synchronous mode transitions

AUTOSAR4: Driver handle the mode transition in asynchronous way afterwards, so no additional handling is necessary.

AUTOSAR3: Higher layer has to handle this (see below).

Loop Name / source	Short Description
kCanLoopStart	MICROSAR3: - Used while transition in mode 'START'. - Call context: Can_SetControllerMode() - There is a delay from writing to a command register until the update of the related status register bits due to clock domain crossing (Host and CAN clock). Therefore the programmer has to assure that the previous value written to INIT has been accepted. - If the loop cancels try to recall Can_SetControllerMode(). MICROSAR4: Used for short time mode transition blocking (short synchronous timeout). Same value for kCanLoopStart, kCanLoopStop, kCanLoopSleep and kCanLoopWakeup. No Issue when timeout occurs.
kCanLoopStop	MICROSAR3: - Used while transition in mode 'STOP'. - Call context: Can_SetControllerMode() - There is a delay from writing to a command register until the update of the related status register bits due to clock domain crossing (Host and CAN clock). Therefore the programmer has to assure that the previous value written to INIT has been accepted. - If the loop cancels try to recall Can_SetControllerMode(). MICROSAR4: Used for short time mode transition blocking (short synchronous timeout). Same value for kCanLoopStart, kCanLoopStop, kCanLoopSleep and kCanLoopWakeup. 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_InitController(), or a Bus-Wakeup event happen. The CAN Driver verifies that no used mailboxes are corrupt. A mailbox is considered corrupt if a predefined pattern is written to the appropriate mailbox registers and the read operation does not return the expected pattern. If a corrupt mailbox is found the function Appl_CanCorruptMailbox() is called. This function tells the application which mailbox is corrupt.

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

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

4.8.5 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_RamCheckExecute()
```

Callouts to notify corrupt mailboxes or controllers:

```
CanIf_RamCheckCorruptController(),CanIf_RamCheckCorruptMailbox()
```

API to re-activate the mailbox or controller again:

```
Can_RamCheckEnableMailbox(),Can_RamCheckEnableController()
```

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

4.9 Common CAN

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

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

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

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

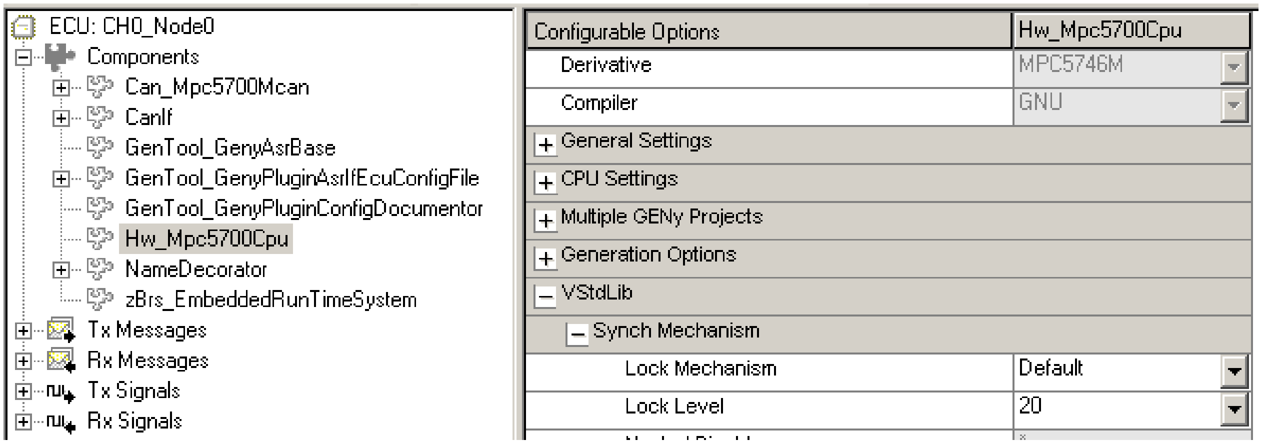
Reason: Upper layers only know one Controller for this 2 hardware channel Common CAN and therefore only one Transceiver can be handled.

4.10 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.

If the application code applies the VStdlib delivered by Vector and it intends to enable/disable the global interrupt using different, then it has configure the “lock level” within GENy accordingly:



Configurable Options	Hw_Mpc5700Cpu
Derivative	MPC5746M
Compiler	GNU
Lock Mechanism	Default
Lock Level	20

Please note that furthermore the dedicated processor core has to be selected accordingly.

By default core – 0 is used. If you decide to take another core, then you have to overwrite the pre-processor macro by using a user-configuration file.

Example (appropriate for SPC58xx derivatives):

```
#define PPC_INTC_CPR  (*(volatile uint32*)  0xFC040000  - with INTC_0 used,
#define PPC_INTC_CPR  (*(volatile uint32*)  0xF4044000  - with INTC_1 used,
```

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.

4.10.1 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.

4.10.2 Not supported

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

The filtering of High Priority messages is not supported.

No Range Filters are supported.

5 Integration

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

5.1 Scope of Delivery

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

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

5.1.1 Static Files

File Name	Description
(+) Can_Local.h	This is an internal header file which should not be included outside this module
(+) Can.c	This is the source file of the CAN. It contains the implementation of CAN module functionality.
(+) Can.lib	This is the library build out of Can.c, Can.h and Can_Local.h
Can.h	This is the header file of the CAN module (include API declaration)
Can_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_Cfg.h	Generated header file, contains some type, prototype and pre-compile settings
Can_Lcfg.c	Generated file contains link time settings.
Can_PBcfg.c	Generated file contains post build settings.
Can_DrvGeneralTypes.h	Generated file contains CAN Driver part of Can_GeneralTypes.h (supported by Integrator)

Table 5-2 Generated files

5.2 Include Structure

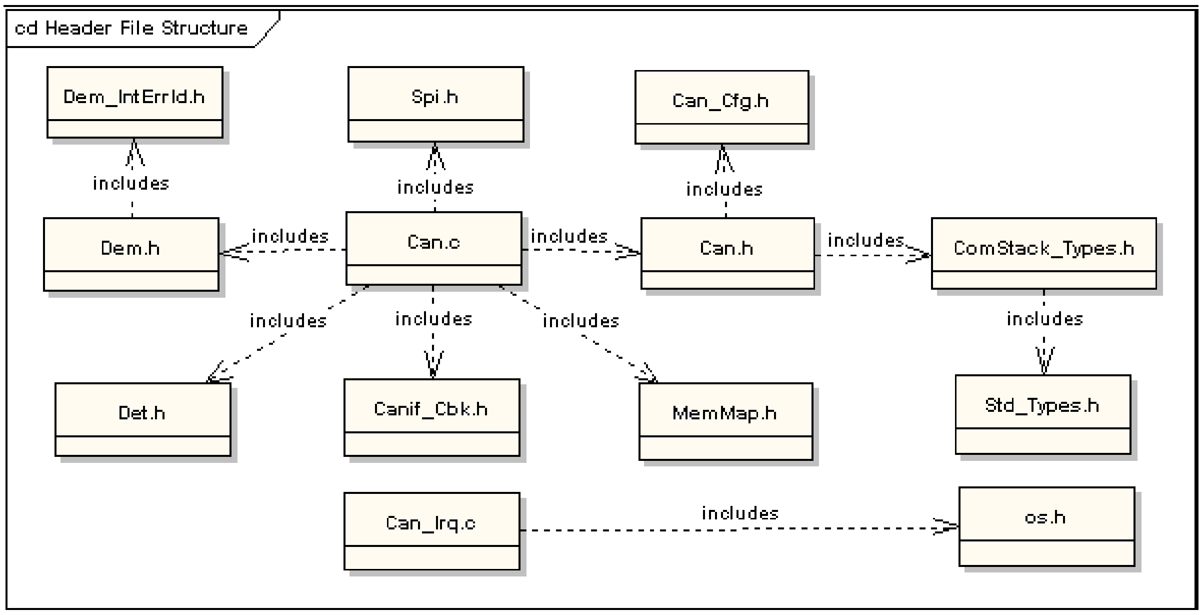


Figure 5-1 Include Structure (AUTOSAR)

Deviation from AUTOSAR specification:

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

5.3 Critical Sections

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

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

Critical Section Define	Description
CAN_EXCLUSIVE_AREA_0	CanNestedGlobalInterruptDisable/Restore() is used within Can_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_EXCLUSIVE_AREA_1	Using inside Can_DisableControllerInterrupts() and Can_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_Disable/EnableControllerInterrupts() are called within context with lower or equal priority than CAN interrupt.
CAN_EXCLUSIVE_AREA_2	Using inside Can_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 does not call function reentrant.
CAN_EXCLUSIVE_AREA_3	Using 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_Write() within.
CAN_EXCLUSIVE_AREA_4	Using 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_EXCLUSIVE_AREA_5	Using 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_EXCLUSIVE_AREA_6	Using inside Can_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

Table 5-3 Critical Section Codes

5.4 Compiler Abstraction and Memory Mapping

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

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

Memory Mapping Sections \ *Compiler Abstraction Definitions*	*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_START_SEC_CODE CAN_STOP_SEC_CODE	■												
CAN_START_SEC_STATIC_CODE CAN_STOP_SEC_STATIC_CODE		■											
CAN_START_SEC_CONST_8BIT CAN_STOP_SEC_CONST_8BIT			■										
CAN_START_SEC_CONST_16BIT CAN_STOP_SEC_CONST_16BIT			■										
CAN_START_SEC_CONST_32BIT CAN_STOP_SEC_CONST_32BIT			■										
CAN_START_SEC_CONST_UNSPECIFIED CAN_STOP_SEC_CONST_UNSPECIFIED			■										
CAN_START_SEC_PBCFG CAN_STOP_SEC_PBCFG				■									
CAN_START_SEC_PBCFG_ROOT CAN_STOP_SEC_PBCFG_ROOT				■									
CAN_START_SEC_VAR_NOINIT_UNSPECIFIED CAN_STOP_SEC_VAR_NOINIT_UNSPECIFIED					■								
CAN_START_SEC_VAR_INIT_UNSPECIFIED CAN_STOP_SEC_VAR_INIT_UNSPECIFIED						■							
CAN_START_SEC_VAR_PBCFG CAN_STOP_SEC_VAR_PBCFG							■						
CAN_START_SEC_CODE_APPL CAN_STOP_SEC_CODE_APPL											■		

Table 5-4 Compiler abstraction and memory mapping

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

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

CAN_CODE: used for CAN module code.

CAN_STATIC_CODE: used for CAN module local code.

CAN_CONST: used for CAN module constants.

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

CAN_VAR_*: used for CAN module variables.

CAN_INT_CTRL: is used to access the CAN interrupt controls.

CAN_REG_CANCELL: is used to access the CAN cell itself.

CAN_RX_TX_DATA: access to CAN Data buffers.

CAN_APPL_*: access to higher layers.

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. The following table shows the corresponding assignment for the derivative RH850 P1X-C.

Critical Section Define	Description
TTCAN_0, BaseAddress: 0xFFD30000 CanIsr_1	CanIsr_0
MCAN_0, BaseAddress: 0xFFEF0000 CanIsr_1	CanIsr_1
MCAN_1, BaseAddress: 0xFFD31000 CanIsr_2	CanIsr_2
MCAN_2, BaseAddress: 0xFFEF1000 CanIsr_3	CanIsr_3

Table 5-5 Hardware Controller – Interrupt Functions

6.2 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, 3.1.0, 3.2.0, 3.2.1
18	Tx FIFO message sequence inversion. Not considered by the CAN Driver, Tx Fifo is not supported.	

Table 6-1 MCAN Errata

6.3 Platform Errata

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

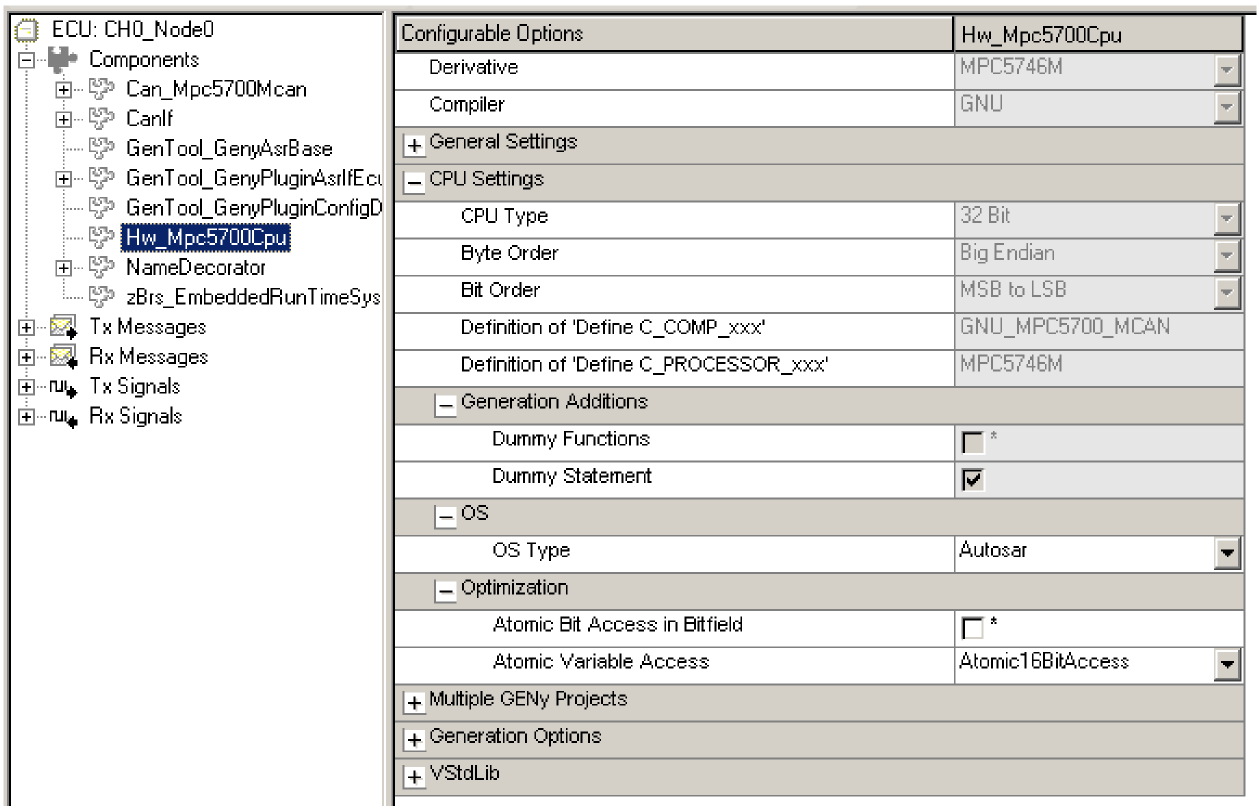
7 API Description

7.1 Interrupt Service Routines provided by CAN

In the decription below the “*platform*” placeholder stands for one of the given platforms (e.g. Mpc5700, Spc5800, Traveo, Rh850, Tricore, Atmel SAM …)

Depending on the settings in Tools component Hw_<*platform*>Cpu, the interrupt routine is given by the driver or by Operating System. (Selection below, not MICROSAR403)

Example for Mpc5700 platform:



- ECU: CH0_Node0
  - Components
    - Can_Mpc5700Mcan
    - CanIf
    - GenTool_GenyAsrBase
    - GenTool_GenyPluginAsrIfEcu
    - GenTool_GenyPluginConfigD
    - Hw_Mpc5700Cpu
    - NameDecorator
    - zBrs_EmbeddedRunTimeSys
  - Tx Messages
  - Rx Messages
  - Tx Signals
  - Rx Signals

Configurable Options	Hw_Mpc5700Cpu
Derivative	MPC5746M
Compiler	GNU
+ General Settings	
– CPU Settings	
CPU Type	32 Bit
Byte Order	Big Endian
Bit Order	MSB to LSB
Definition of 'Define C_COMP_xxx'	GNU_MPC5700_MCAN
Definition of 'Define C_PROCESSOR_xxx'	MPC5746M
– Generation Additions	
Dummy Functions	☐ *
Dummy Statement	✓
– OS	
OS Type	Autosar
– Optimization	
Atomic Bit Access in Bitfield	☐ *
Atomic Variable Access	Atomic16BitAccess
+ Multiple GENy Projects	
+ Generation Options	
+ VStdLib	

Figure 7-1 Select OS Type

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

7.1.1 OSEK (OS)

This means to include osek.h.

Switch: V_OSTYPE_OSEK

7.1.2 AutoSar (OS)

Os.h header file is used.

Switch: V_OSTYPE_AUTOSAR

7.1.3 None (OS)

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

7.1.4 Type of Interrupt Function

See Chapter “Component Settings” for configuration aspects.

- Category 2 (only for OSEK OS or AUTOSAR OS):
  A macro “ISR(CanIsr_x)” will be used to declare ISR function call. The name given as parameter for interrupt naming (x = Physical CAN Channel number). For macro definition see OS specification. The OS has full control of the ISR.
  switch: C_ENABLE_OSEK_OS_INTCAT2
- Category 1:
  Using OS with category 1 interrupts need an Interface layer handling these interrupts in task context like defined in BSW00326 (AUTOSAR_SRS_General).
  switch: C_DISABLE_OSEK_OS_INTCAT2
- Void-Void Interrupt Function:
  Like in Category 1 the Interrupt is not handled by OS and the ISR is declared as void ISR(void) and has to be called by interrupt controller in case of an CAN interrupt.
  switch: C_ENABLE_ISRVOID

7.1.5 CAN ISR API

Prototype	
`void **CanIsr_<x>**(void);`	
Parameter	
---	---
Return code	
---	---
Functional Description	
Handles interrupts of hardware channel <x> for Rx, Tx, BusOff events.	
Particularities and Limitations	
> Number of available functions depends on used MCU derivative. > The functions are not designated as interrupt functions. If it is necessary to save/restore all general purpose registers and to use a different “return from interrupt” instruction the application code has to implement the compiler specific pragma (e.g. for Wind River™ DIAB™: #pragma interrupt CanIsr_x).	

Table 7-1 MCAN CanIsr_<x>

7.2 Services provided by CAN

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

7.2.1 Can_InitMemory

Prototype
`void **Can_InitMemory** (void)`

Parameter	
-	
Return code	
void	-
Functional Description	
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_Init()".	
Particularities and Limitations	
Called by Application.	
	Caution None AUTOSAR API
Call context	
> Should be called while power on initialization before "Can_Init()" on task level. > This function is Synchronous > This function is Non-Reentrant > Availability: Always	

Table 7-2 Can_InitMemory

7.2.2 Can_Init

Prototype	
`void Can_Init (Can_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_InitController()" but after call of "Can_InitMemory()". > This function is Synchronous > This function is Non-Reentrant > Availability: Always

Table 7-3 Can_Init

7.2.3 Can_InitController

Prototype	
`void` **`Can_InitController`** `(uint8 Controller, Can_ControllerBaudrateConfigPtrType Config)`	
Parameter	
Controller [in]	Number of controller
Config [in]	Pointer to baud rate configuration structure
Return code	
void	-
Functional Description	
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_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_InitController

7.2.4 Can_InitController

Prototype
`void` **`Can_InitController`** `(uint8 Controller, Can_ControllerConfigPtrType ControllerConfigPtr)`

Parameter	
Controller [in]	Number of controller
Config [in]	Pointer to the configuration data structure.
Return code	
void	-
Functional Description	
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_Init()". > Must not be called while in "Sleep Mode". > This function is Synchronous > This function is Non-Reentrant > Availability: MICROSAR3 only	

Table 7-5 Can_InitController

7.2.5 Can_ChangeBaudrate

Prototype	
`Std_ReturnType` **`Can_ChangeBaudrate`** `(uint8 Controller, const uint16 Baudrate)`	
Parameter	
Controller [in]	Number of controller to be changed
Baudrate [in]	Baud rate to be set
Return code	
Std_ReturnType	> 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_Init()". > This function is Synchronous > This function is Non-Reentrant > Availability: MICROSAR403 only & if "CanChangeBaudrateApi" is activated or "CanSetBaudrateApi" is de-activated.

Table 7-6 Can_ChangeBaudrate

7.2.6 Can_CheckBaudrate

Prototype	
`Std_ReturnType` **`Can_CheckBaudrate`** `(uint8 Controller, const uint16 Baudrate)`	
Parameter	
Controller [in]	Number of controller to be checked
Baudrate [in]	Baud rate to be checked
Return code	
Std_ReturnType	> 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_CHANGE_BAUDRATE_API == STD_ON")	

Table 7-7 Can_CheckBaudrate

7.2.7 Can_SetBaudrate

Prototype	
`Std_ReturnType` **`Can_SetBaudrate`** `(uint8 Controller, uint16 BaudRateConfigID)`	
Parameter	
Controller [in]	Number of controller to be set
BaudRateConfigID [in]	Identity of the configured baud rate (available as Symbolic Name)

Return code	
Std_ReturnType	> 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_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_SET_BAUDRATE_API == STD_ON")	

Table 7-8 Can_SetBaudrate

7.2.8 Can_InitStruct

Prototype	
`void Can_InitStruct (uint8 Controller, uint8 Index)`	
Parameter	
Controller [in]	Number of the controller to be changed
Index [in]	Index of the initialization structure to be used for baud rate and mask settings
Return code	
void	-
Functional Description	
Set content of the initialization structure (before calling "Can_InitController()"). Service function to change the initialization structure setup left behind by the Generation Tool. The structure contains information about baud rate and filter settings. Subsequent "Can_InitController()" must be called to activate these settings.	
Particularities and Limitations	
Called by Application. "Can_Init" was called.	
⚠	**Caution** None AUTOSAR API
Call context	
> Call this function between calling "Can_Init()" and "Can_InitController()".	

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

Table 7-9 Can_InitStruct

7.2.9 Can_GetVersionInfo

Prototype	
void **Can_GetVersionInfo** (Can_VersionInfoPtrType VersionInfo)	
Parameter	

VersionInfo [out]	Pointer to where to store the version information of the CAN Driver.
	typedef struct {
	uint16 vendorID;
	uint16 moduleID;
	MICROSAR3 only: uint8 instanceID;
	uint8 sw_major_version; (MICROSAR3 only: BCD coded)
	uint8 sw_minor_version; (MICROSAR3 only: BCD coded)
	uint8 sw_patch_version; (MICROSAR3 only: BCD coded)
	} Std_VersionInfoType;

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_VERSION_INFO_API == STD_ON")	

Table 7-10 Can_GetVersionInfo

7.2.10 Can_GetStatus

Prototype	
uint8 **Can_GetStatus** (uint8 Controller)	
Parameter	
Controller [in]	Number of the controller requested for status information

Return code	
uint8	> CAN_STATUS_START > CAN_STATUS_STOP (Bit coded status information) > CAN_STATUS_INIT > CAN_STATUS_INCONSISTENT, > CAN_DEACTIVATE_CONTROLLER (only with "CanRamCheck" active) > CAN_STATUS_WARNING > CAN_STATUS_PASSIVE > CAN_STATUS_BUSOFF > CAN_STATUS_SLEEP
Functional Description	
Delivers the status of the hardware. Only one of the status bits CAN_STATUS_SLEEP/ STOP/ START/ BUSOFF/ PASSIVE/ WARNING is set. The CAN_STATUS_INIT bit is always set if a controller is initialized. CAN_STATUS_SLEEP has the highest and CAN_STATUS_WARNING the lowest priority. CAN_STATUS_INCONSISTENT will be set if one Common CAN channel. is not "Stop" or "Sleep". CAN_DEACTIVATE_CONTROLLER is set in case the "CanRamCheck" detected an Issue. "status" can be analyzed using the provided API macros: CAN_HW_IS_OK(status): return "true" in case no warning, passive or bus off occurred. CAN_HW_IS_WARNING(status): return "true" in case of waning status. CAN_HW_IS_PASSIVE(status): return "true" in case of passive status. CAN_HW_IS_BUSOFF(status): return "true" in case of bus off status (may be already false in Notification). CAN_HW_IS_WAKEUP(status): return "true" in case of not in Sleep Mode. CAN_HW_IS_SLEEP(status): return "true" in case of Sleep Mode. CAN_HW_IS_STOP(status): return "true" in case of Stop Mode. CAN_HW_IS_START(status): return "true" in case of not in Stop Mode. CAN_HW_IS_INCONSISTENT(status): return "true" in case of an inconsistency between two common CAN channels.	
Particularities and Limitations	
Called by network management or Application.	
	Caution None AUTOSAR API
Call context	
> This function is Synchronous > This function is Non-Reentrant > Availability: "CanGetStatus" is activated ("CAN_GET_STATUS == STD_ON")	

Table 7-11 Can_GetStatus

7.2.11 Can_SetControllerMode

Prototype	
`Can_ReturnType` **`Can_SetControllerMode`** `(uint8 Controller, Can_StateTransitionType Transition)`	
Parameter	
Controller [in]	Number of the controller to be set
Transition [in]	Requested transition to destination mode
Return code	
Can_ReturnType	> 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 7-12 Can_SetControllerMode

7.2.12 Can_ResetBusOffStart

Prototype	
`void` **`Can_ResetBusOffStart`** `(uint8 Controller)`	
Parameter	
Controller [in]	Number of the controller
Return code	
void	-
Functional Description	
This is a compatibility function (for a CANbedded protocol stack) used during the start of the Bus Off handling to remove the Bus Off state.	
Particularities and Limitations	
Called by CAN Driver.	

	Caution None AUTOSAR API
Call context	
> Called while BusOff event handling (Polling or Interrupt context). > This function is Synchronous > This function is Non-Reentrant > Availability: Always	

Table 7-13 Can_ResetBusOffStart

7.2.13 Can_ResetBusOffEnd

Prototype	
`void Can_ResetBusOffEnd (uint8 Controller)`	
Parameter	
Controller [in]	Number of the controller
Return code	
void	-
Functional Description	
This is a compatibility function (for a CANbedded protocol stack) used during the end of the Bus Off handling to remove the Bus Off state.	
Particularities and Limitations	
Called by CAN Driver.	
	Caution None AUTOSAR API
Call context	
> Called inside "Can_SetControllerMode()" while Start transition. > This function is Synchronous > This function is Non-Reentrant > Availability: Always	

Table 7-14 Can_ResetBusOffEnd

7.2.14 Can_Write

Prototype	
`Can_ReturnType Can_Write (Can_HwHandleType Hth, Can_PduInfoPtrType PduInfo)`	
Parameter	
Hth [in]	Handle of the mailbox intended to send the message

PduInfo [in]	Information about the outgoing message (ID, dataLength, data)
Return code	
Can_ReturnType	> 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 7-15 Can_Write

7.2.15 Can_CancelTx

Prototype	
`void Can_CancelTx (Can_HwHandleType Hth, PduIdType PduId)`	
Parameter	
Hth [in]	Handle of the mailbox intended to be cancelled.
PduId [in]	Pdu identifier
Return code	
void	-
Functional Description	
Cancel the TX message in the hardware buffer (if possible) or mark the message as not to be confirmed in case of the cancellation is unsuccessful.	
Particularities and Limitations	
Called by CanTp or Application.	
⚠	**Caution** None AUTOSAR API
Call context	
> Called by CanTp or Application. > This function is Synchronous	

> This function is Non-Reentrant > Availability: Always

Table 7-16 Can_CancelTx

7.2.16 Can_SetMirrorMode

Prototype	
`void Can_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_GenericPreCopy/Confirmation function ON or OFF.	
Particularities and Limitations	
Configuration Variant(s): C_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 7-17 Can_SetMirrorMode

7.2.17 Can_SetSilentMode

Prototype	
`Std_ReturnType Can_SetSilentMode (uint8 Controller, Can_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_SILENT_MODE == STD_ON
Call context
> TASK > This function is Synchronous > This function is Non-Reentrant

Table 7-18 Can_SetSilentMode

7.2.18 Can_CheckWakeup

Prototype	
`Std_ReturnType` **`Can_CheckWakeup`** `(uint8 Controller)`	
Parameter	
Controller [in]	Number of the controller to be checked for Wake Up events.
Return code	
Std_ReturnType	> 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_CheckWakeup", in AR3.x named "Can_Cbk_CheckWakeup" (Name mapped by define)	

Table 7-19 Can_CheckWakeup

7.2.19 Can_DisableControllerInterrupts

Prototype
`void` **`Can_DisableControllerInterrupts`** `(uint8 Controller)`

Parameter	
Controller [in]	Number of the CAN controller to disable interrupts for.
Return code	
void	-
Functional Description	
Service function to disable the CAN interrupt for the given controller (e.g. due to data security reasons).	
Particularities and Limitations	
Called by SchM. Must not be called while CAN controller is in Sleep Mode.	
Call context	
> Called within Critical Area handling or out of Application code. > This function is Synchronous > This function is Non-Reentrant > Availability: Always	

Table 7-20 Can_DisableControllerInterrupts

7.2.20 Can_EnableControllerInterrupts

Prototype	
`void Can_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 7-21 Can_EnableControllerInterrupts

7.2.21 Can_MainFunction_Write

Prototype	
`void Can_MainFunction_Write (void)`	
Parameter	
-	
Return code	
void	-
Functional Description	
Service function to poll TX events (confirmation, cancellation) for all controllers and all TX mailboxes to accomplish the TX confirmation handling (like CanInterface notification).	
Particularities and Limitations	
Called by SchM. Must not interrupt the call of "Can_Write()".	
Call context	
> Called within cyclic TX task. > This function is Synchronous > This function is Non-Reentrant > Availability: Always	

Table 7-22 Can_MainFunction_Write

7.2.22 Can_MainFunction_Read

Prototype	
`void Can_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 7-23 Can_MainFunction_Read

7.2.23 Can_MainFunction_BusOff

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

Table 7-24 Can_MainFunction_BusOff

7.2.24 Can_MainFunction_Wakeup

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

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

Table 7-25 Can_MainFunction_Wakeup

7.2.25 Can_MainFunction_Mode

Prototype	
`void Can_MainFunction_Mode (void)`	
Parameter	
-	
Return code	
void	-
Functional Description	
Service function to poll Mode changes over all controllers. (This is handled asynchronous if not accomplished in "Can_SetControllerMode()").	
Particularities and Limitations	
Called by SchM.	
Call context	
> Called within cyclic mode change task. > This function is Synchronous > This function is Non-Reentrant > Availability: MICROSAR4x only	

Table 7-26 Can_MainFunction_Mode

7.2.26 Can_RamCheckExecute

Prototype	
`void Can_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_RAM_CHECK == CAN_EXTENDED CREQ-106641
Call context
> TASK > This function is Synchronous > This function is Non-Reentrant

Table 7-27 Can_RamCheckExecute

7.2.27 Can_RamCheckEnableMailbox

Prototype	
`void Can_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_RamCheckEnableController() to activate mailbox and controller. Configuration Variant(s): CAN_RAM_CHECK == CAN_EXTENDED CREQ-106641	
Call context	
> TASK > This function is Synchronous > This function is Non-Reentrant	

Table 7-28 Can_RamCheckEnableMailbox

7.2.28 Can_RamCheckEnableController

Prototype	
`void Can_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_RAM_CHECK == CAN_EXTENDED CREQ-106641	
Call context	
> TASK > This function is Synchronous > This function is Non-Reentrant	

Table 7-29 Can_RamCheckEnableController

7.2.29 Appl_GenericPrecopy

Prototype	
`Can_ReturnType` **`Appl_GenericPrecopy`** `(uint8 Controller, Can_IdType ID, uint8 DataLength, Can_DataPtrType DataPtr)`	
Parameter	
Controller [in]	Controller which received the message
ID [in]	ID of the received message (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_GENERIC_PRECOPY == STD_ON").

Table 7-30 Appl_GenericPrecopy

7.2.30 Appl_GenericConfirmation

Prototype	
`Can_ReturnType` **`Appl_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_GENERIC_CONFIRMATION == STD_ON") & "CanIfTransmitBuffer" activated (in CanInterface).	

Table 7-31 Appl_GenericConfirmation

7.2.31 Appl_GenericConfirmation

Prototype	
`Can_ReturnType` **`Appl_GenericConfirmation`** `(uint8 Controller, Can_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_GENERIC_CONFIRMATION == CAN_API2").	

Table 7-32 Appl_GenericConfirmation

7.2.32 Appl_GenericPreTransmit

Prototype	
`void` **`Appl_GenericPreTransmit`** `(uint8 Controller, Can_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_Write()". > This function is Synchronous > This function is Non-Reentrant > Availability: "CanGenericPretransmit" is activated ("CAN_GENERIC_PRETRANSMIT == STD_ON").

Table 7-33 Appl_GenericPreTransmit

7.2.33 ApplCanTimerStart

Prototype	
`void ApplCanTimerStart (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_HW_LOOP_SUPPORT_API == STD_ON").	

Table 7-34 ApplCanTimerStart

7.2.34 ApplCanTimerLoop

Prototype	
`Can_ReturnType` **`ApplCanTimerLoop`** `(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_HW_LOOP_SUPPORT_API == STD_ON").	

Table 7-35 ApplCanTimerLoop

7.2.35 ApplCanTimerEnd

Prototype	
`void` **`ApplCanTimerEnd`** `(CanChannelHandle Controller, uint8 source)`	
Parameter	
Controller [in]	Number of the controller on which the hardware observation takes place. (only if not using "Optimize for one controller")
source [in]	Source for the hardware observation (see chapter Hardware Loop Check / Timeout Monitoring).

Return code	
void	-
Functional Description	
Service function to to end an observation timer (see chapter Hardware Loop Check / Timeout Monitoring).	
Particularities and Limitations	
Called by CAN Driver.	
⚠	**Caution** None AUTOSAR API
Call context	
> For context information please refer to chapter "Hardware Loop Check". > This function is Synchronous > This function is Non-Reentrant > Availability: "CanHardwareCancelByAppl" is activated ("CAN_HW_LOOP_SUPPORT_API == STD_ON").	

Table 7-36 ApplCanTimerEnd

7.2.36 ApplCanInterruptDisable

Prototype	
`void` **`ApplCanInterruptDisable`** `(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_DisableControllerInterrupts()". > This function is Synchronous > This function is Non-Reentrant	

> Availability: "CanInterruptLock" is set to APPL or BOTH ("CAN_INTLOCK == CAN_APPL" or "CAN_INTLOCK == CAN_BOTH").

Table 7-37 ApplCanInterruptDisable

7.2.37 ApplCanInterruptRestore

Prototype	
`void **ApplCanInterruptRestore** (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_EnableControllerInterrupts()". > This function is Synchronous > This function is Non-Reentrant > Availability: "CanInterruptLock" is set to APPL or BOTH ("CAN_INTLOCK == CAN_APPL" or "CAN_INTLOCK == CAN_BOTH").	

Table 7-38 ApplCanInterruptRestore

7.2.38 Appl_CanOverrun

Prototype	
`void **Appl_CanOverrun** (uint8 Controller)`	
Parameter	
Controller [in]	Number of the controller for which the overrun was detected.
Return code	
void	-

Functional Description
This function will be called when an overrun is detected for a BasicCAN mailbox. Alternatively, a DET call can be selected instead of ("CanOverrunNotification" is set to "DET").
Particularities and Limitations
Called by CAN Driver.
⚠ **Caution** None AUTOSAR API
Call context
> Called within CAN message reception or error detection context (Polling or Interrupt). > This function is Synchronous > This function is Non-Reentrant > Availability: "CanOverrunNotification" set to APPL ("CAN_OVERRUN_NOTIFICATION == CAN_APPL").

Table 7-39 Appl_CanOverrun

7.2.39 Appl_CanFullCanOverrun

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

Table 7-40 Appl_CanFullCanOverrun

7.2.40 Appl_CanCorruptMailbox

Prototype	
`void Appl_CanCorruptMailbox (uint8 Controller, Can_HwHandleType hwObjHandle)`	
Parameter	
Controller [in]	Number of the controller for which the check failed.
hwObjHandle [in]	Hardware handle of the defect mailbox.
Return code	
void	-
Functional Description	
This function will notify the application (during "Can_InitController()") about a defect mailbox within the CAN cell.	
Particularities and Limitations	
Called by CAN Driver.	
⚠	**Caution** None AUTOSAR API
Call context	
> Call within controller initialization. > This function is Synchronous > This function is Non-Reentrant > Availability: "CanRamCheck" set to "MailboxNotifiation" ("CAN_RAM_CHECK == CAN_NOTIFY_MAILBOX").	

Table 7-41 Appl_CanCorruptMailbox

7.2.41 Appl_CanRamCheckFailed

Prototype	
`uint8 Appl_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_DEACTIVATE_CONTROLLER - deactivate the controller > CAN_ACTIVATE_CONTROLLER - activate the controller
Functional Description	
This function will notify the application (during "Can_InitController()") about a defect CAN controller due to a previous failed mailbox check.	

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

Table 7-42 Appl_CanRamCheckFailed

7.2.42 ApplCanInitPostProcessing

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

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 macros are 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').

- **CanCRelReg**(controller): the CREL register of the specified CAN channel can be read according to the register definition (see Hardware Manufacturer Document in ch. 2).

- **CanGetMcanRevision**(controller): the CREL register of the specified CAN channel is read and the release information (Version, Step and Sub Step) is returned as a vuint16 type.

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_ENABLE_INIT_POST_PROCESS' is defined via a user-config file.

Table 7-43 ApplCanInitPostProcessing

7.3 Services used by CAN

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

Component	API
DET	**Det_ReportError** (see "Development Error Reporting")
DEM	**Dem_ReportErrorStatus** (see "Production Code Error Reporting")
EcuM	**EcuM_CheckWakeup** This function is called when Wakeup over CAN bus occur. **EcuM_GeneratorCompatibilityError** This function is called during the initialization, of the CAN Driver if the Generator Version Check or the CRC Check fails. (see [5])
Application (optional non AUTOSAR)	**Appl_GenericPrecopy** **Appl_GenericConfirmation** **Appl_GenericPreTransmit** **ApplCanTimerStart/Loop/End** **Appl_CanRamCheckFailed, Appl_CanCorruptMailbox** **ApplCanInterruptDisable/Restore** **Appl_CanOverrun** For detailed description see Chapter 7.2

CANIF	**CanIf_CancelTxNotification** (non AUTOSAR) A special Software cancellation callback only used within Vector CAN Driver CAN Interface bundle. **CanIf_TxConfirmation** Notification for a successful transmission. (see [4]) **CanIf_CancelTxConfirmation** Notification for a successful Tx cancellation. (see [4]) **CanIf_RxIndication** Notification for a message reception. (see [4]) **CanIf_ControllerBusOff** Bus Off notification function. (see [4]) **CanIf_ControllerModeIndication** MICROSAR4x only: Notification for mode 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.

Table 7-44 Services used by the CAN

8 Configuration

For CAN driver the attributes can be configured with configuration tool “CFG5”. The CAN Driver supports pre-compile, link-time and post-build configuration. For post-build systems, re-flashing the generated data can change some configuration settings. For post-build and link-time configurations pre-compile settings are configured at compile time and therefore unchangeable at link or post-build time.

The following parameters are set by CFG5 configuration (see chapter “Configuration with GENy”).

8.1 Pre-Compile Parameters

Settings have to be available before compilation:

- Version API (Can_GetVersionInfo() activation)
```
#define CAN_VERSION_INFO_API         STD_ON/STD_OFF
```
- DET (development error detection)
```
#define CAN_DEV_ERROR_DETECT         STD_ON/STD_OFF
```
- Hardware Loop Check (timeout monitoring)
```
#define CAN_HARDWARE_CANCELLATION    STD_ON/STD_OFF
```
- Polling modes: Tx confirmation, Reception, Wakeup, BusOff
```
#define CAN_TX_PROCESSING   CAN_INTERRUPT/CAN_POLLING
#define CAN_RX_PROCESSING CAN_INTERRUPT/CAN_POLLING
#define CAN_BUSOFF_PROCESSINGCAN_INTERRUPT/CAN_POLLING
#define CAN_WAKEUP_PROCESSINGCAN_INTERRUPT/CAN_POLLING
#define CAN_INDIVIDUAL_PROCESSING    STD_ON/STD_OFF
```
- Multiplexed Tx (external PIA – by usage of multiple Tx mailboxes)
```
#define CAN_MULTIPLEXED_TRANSMISSION   STD_ON/STD_OFF
```
- Configuration Variant (define the configuration type when using post build variant)
```
#define CAN_ENABLE_SELECTABLE_PB
```
- Use Generic Precopy Function (None AUTOSAR feature)
```
#define CAN_GENERIC_PRECOPY          STD_ON/STD_OFF
```
- Use Generic Confirmation Function (None AUTOSAR feature)
```
#define CAN_GENERIC_CONFIRMATION      STD_ON/STD_OFF
```
- Use Rx Queue Function (None AUTOSAR feature)
```
#define CAN_RX_QUEUE               STD_ON/  (not supported)
                                   STD_OFF
```
- Used ID type (standard/extended or mixed ID format)
```
#define CAN_EXTENDED_ID             STD_ON/STD_OFF
#define CAN_MIXED_ID                STD_ON/STD_OFF
```
- Usage of Rx and Tx Full and BasicCAN objects (deactivate only when not using and to save ROM and runtime consumption)
```
#define CAN_RX_FULLCAN_OBJECTS         STD_ON/STD_OFF
#define CAN_TX_FULLCAN_OBJECTS         STD_ON/STD_OFF
#define CAN_RX_BASICCAN_OBJECTS        STD_ON/STD_OFF
```
- Use Multiple BasicCAN objects
```
#define CAN_MULTIPLE_BASICCAN           STD_ON/STD_OFF
```

> Optimizations
```
#define CAN_ONE_CONTROLLER_OPTIMIZATION STD_ON/STD_OFF
#define CAN_DYNAMIC_FULLCAN_ID          STD_ON/STD_OFF
```

> Usage of nested CAN interrupts
```
#define CAN_NESTED_INTERRUPTS           STD_ON/STD_OFF
```

> Use Multiple ECU configurations
```
#define CAN_MULTI_ECU_CONFIG             STD_ON/STD_OFF
```

> Use RAM Check (verify mailbox buffers)
```
#define CAN_RAM_CHECK         CAN_NONE/
                                        CAN_NOTIFY_ISSUE/
                                        CAN_NOTIFY_MAILBOX
```

> Use Overrun detection
```
#define CAN_OVERRUN_NOTIFICATION     CAN_NONE/
                                              CAN_DET/
                                              CAN_APPL
```

> Select MicroSar version
```
#define CAN_MICROSAR_VERSION         CAN_MSR30/
                                              CAN_MSR40
```

> Tx Cancellation of Identical IDs
```
#define CAN_IDENTICAL_ID_CANCELLATION    STD_ON/STD_OFF
```

8.2 Link-Time Parameters

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

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

> Rx Queue size

> Controller mapping (mapping of logical channel to hardware node).

> CAN hardware base address.

8.3 Post-Build Parameters

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

> Amount and usage of FullCAN Rx and Tx mailboxes

> Used database (message information like ID, DLC)

> Filters for BasicCAN Rx mailbox

> Baud-rate settings

> Module Start Address (only for post-build systems: The memory location for re-flashed data has to be defined)

> Configuration ID (only for post-build systems: This number is used to identify the post-build data
> CAN hardware Fifo depth
> CAN hardware clock and bit timing settings

8.4 Configuration with GENy

Please assure that your platform is available with GenTool GENy !

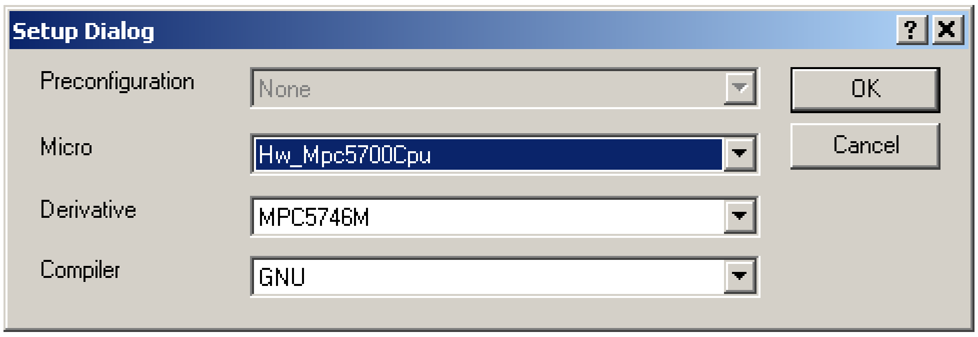
The CAN driver is configured with the help of the configuration tool GENy. To generate communication specific settings, like used message objects e.g. FullCAN Rx mailboxes, baud rate settings and hardware specific settings for CAN communication, the GENy tool is used. This chapter explains the usage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

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

8.4.1 Platform Settings

Platform settings are done in the initial setup dialog:

The example below shows Mpc5700 platform:



Preconfiguration: None
Micro: Hw_Mpc5700Cpu
Derivative: MPC5746M
Compiler: GNU

Figure 8-1 Platform 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.

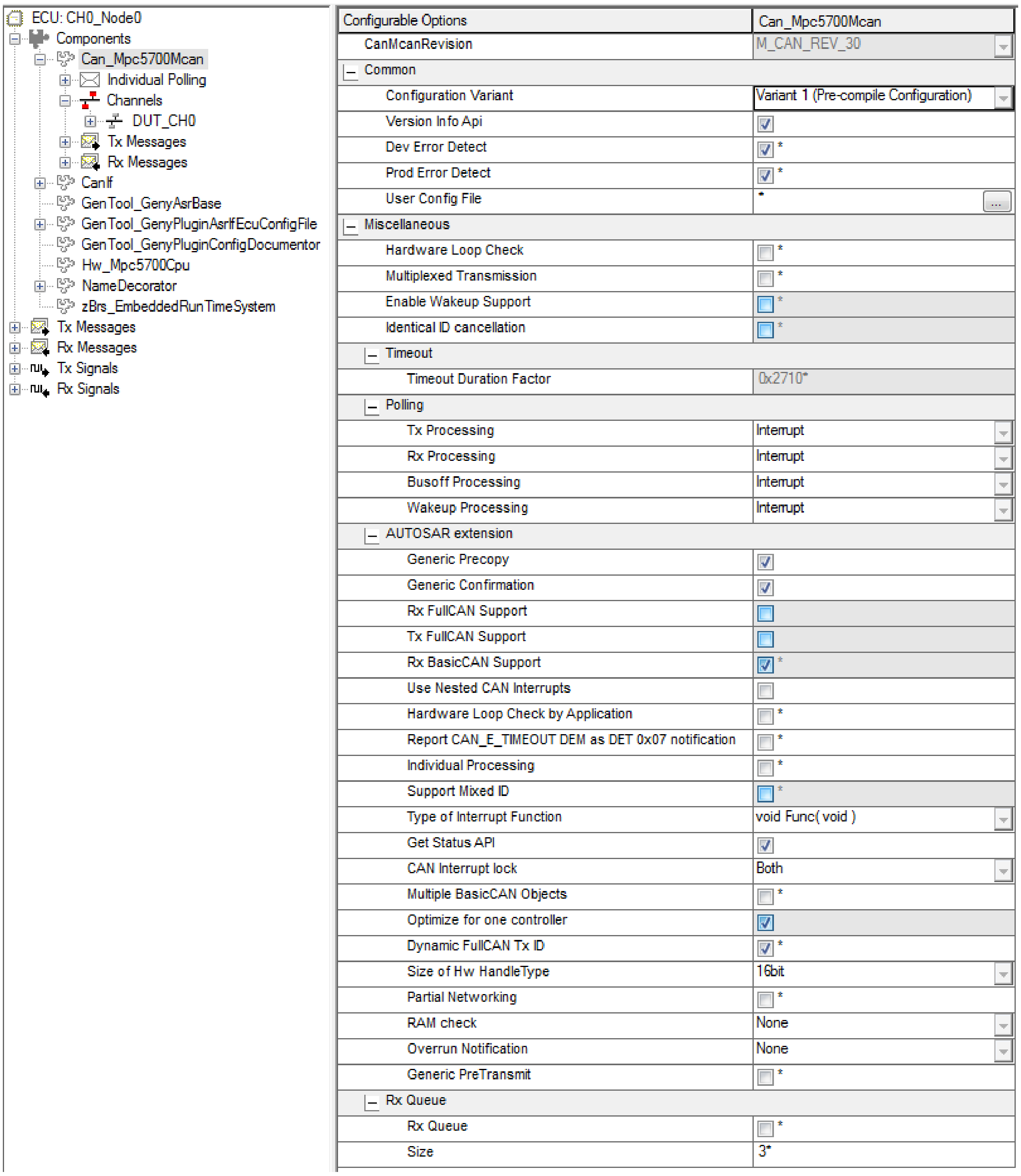
Compiler Selection

Select GNU if compiler HighTec for the PowerPC™ is used.

Select DIAB if compiler DiabData for the PowerPC™ is used
(Wind River Compiler toolkit, formerly known as the Diab Compiler).

Select GHS if compiler Greenhills for the PowerPC™ is used.

8.4.2 Component Settings

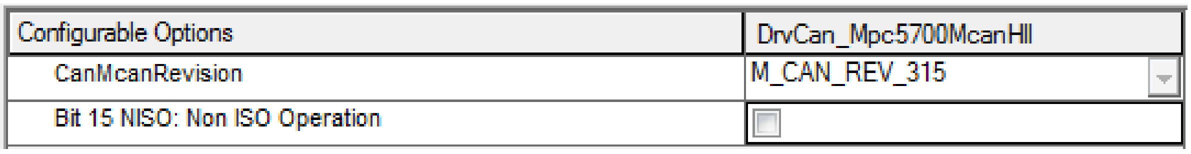


- ECU: CH0_Node0
  - Components
    - Can_Mpc5700Mcan
      - Individual Polling
      - Channels
        - DUT_CH0
      - Tx Messages
      - Rx Messages
    - CanIf
    - GenTool_GenyAsrBase
    - GenTool_GenyPluginAsrIfEcuConfigFile
    - GenTool_GenyPluginConfigDocumentor
    - Hw_Mpc5700Cpu
    - NameDecorator
    - zBrs_EmbeddedRunTimeSystem
  - Tx Messages
  - Rx Messages
  - Tx Signals
  - Rx Signals

Configurable Options	Can_Mpc5700Mcan
CanMcanRevision	M_CAN_REV_30
Common	
Configuration Variant	Variant 1 (Pre-compile Configuration)
Version Info Api	✓
Dev Error Detect	✓ *
Prod Error Detect	✓ *
User Config File	* ...
Miscellaneous	
Hardware Loop Check	☐ *
Multiplexed Transmission	☐ *
Enable Wakeup Support	☐ *
Identical ID cancellation	☐ *
Timeout	
Timeout Duration Factor	0x2710*
Polling	
Tx Processing	Interrupt
Rx Processing	Interrupt
Busoff Processing	Interrupt
Wakeup Processing	Interrupt
AUTOSAR extension	
Generic Precopy	✓
Generic Confirmation	✓
Rx FullCAN Support	☐
Tx FullCAN Support	☐
Rx BasicCAN Support	✓ *
Use Nested CAN Interrupts	☐
Hardware Loop Check by Application	☐ *
Report CAN_E_TIMEOUT DEM as DET 0x07 notification	☐ *
Individual Processing	☐ *
Support Mixed ID	☐ *
Type of Interrupt Function	void Func(void)
Get Status API	✓
CAN Interrupt lock	Both
Multiple BasicCAN Objects	☐ *
Optimize for one controller	✓
Dynamic FullCAN Tx ID	✓ *
Size of Hw HandleType	16bit
Partial Networking	☐ *
RAM check	None
Overrun Notification	None
Generic PreTransmit	☐ *
Rx Queue	
Rx Queue	☐ *
Size	3*

Figure 8-2 Component Settings

8.4.2.1 CAN-FD Non ISO Compatibility



Configurable Options	DrvCan_Mpc5700McanHll
CanMcanRevision	M_CAN_REV_315
Bit 15 NISO: Non ISO Operation	☐

When selecting MCAN Rev. 3.1.5 or later an additional configuration attribute “Non ISO Operation” comes up which allows to select either “ISO 11898-1” or “Bosch CAN-FD Spec. V1.0” oeration of CAN-FD.

8.4.2.2 Attribute Description List

Attribute Name	Configuration Variant	Value Type	Values	Description
Configurable Options				
Can MCAN Revision	pre-compile	Enum	M_CAN_REV_10 M_CAN_REV_20 M_CAN_REV_30 M_CAN_REV_310 M_CAN_REV_315	Select the MCAN Release of your derivative Rel.1, Rel.2, Rel.3, Rel.3, Step1, Sub Step 0, Rel.3, Step1, Sub Step 5, For further revisions (3.2.x) M_CAN_REV_315 can be used
Non ISO Operation	pre-compile	Enum	ISO-11898, Bosch_V10	Select either ISO 11898-1 or Bosch CAN-FD Spec. V1.0 compatibility
Common				
Configuration Variant	pre-compile	Enum	pre-compile, link-time, post-build	Select your project type
Version Info API	pre-compile	Bool	On / Off	Activate function to get version information
Dev Error Detection	pre-compile	Bool	On / Off	Activate to get information about possible conflicts (refer to DET module description)
Prod Error Detection	pre-compile	Bool	On / Off	Activate to get information about possible conflicts in a running system (refer to DEM module description)
User Config File	pre-compile	String	File name and path	This file will be included in the Can_cfg.h for special settings
Miscellaneous				
Hardware Loop Check	pre-compile	Bool	On / Off	Activate to check for hardware problems inside a loop (possible endless loops because of hardware issues will be notified over DEM).

Attribute Name	Configuration Variant	Value Type	Values	Description
Multiplexed Transmission	pre-compile	Bool	On / Off	Activate to get multiple transmit objects for the Tx BasicCAN object (3 send objects instead of 1). This avoids external Id priority inversion.
Wakeup Support	pre-compile	Bool	Off	Activate to Wake up over CAN bus by CAN Driver.
Identical ID cancellation	pre-compile	Bool	On / Off	'Hardware Transmit Cancellation" also cancels messages with the same identifier (newer data will be send) out of hardware mailboxes. Otherwise only lower prior identifiers will be cancelled (older data will be send).
Timeout				
Timeout Duration Factor (MICROSAR3 only)	pre-compile	Integer	Loop counter	Maximum loop count for hardware loop check. Refer to loop descriptions set this value for your needs.
Timeout Duration (MICROSAR4x only)	pre-compile	Float	Time	Specifies the time (in seconds) for the “Hardware Loop Check”.
Timeout Counter ID (MICROSAR401 only)	pre-compile	Float	Time	Specifies the counter ID used in OS for 'Timeout Duration'
Polling				
Tx Processing	pre-compile	Enum	Interrupt / Polling	Activate to handle transmit messages over polling
Rx Processing	pre-compile	Enum	Interrupt / Polling	Activate to handle receive messages over polling
Busoff Processing	pre-compile	Enum	Interrupt / Polling	Activate to handle BusOff event over Polling
Wakeup Processing	pre-compile	Enum	Interrupt / Polling	Activate to handle WakeUp events by Polling Mode.
AUTOSAR extension				
Generic PreCopy	pre-compile	Bool	On / Off	Generic precopy function for all receive messages
Generic Confirmation	pre-compile	Bool	On / Off	Generic Tx confirmation for all transmit messages
Rx FullCAN Support	pre-compile	Bool	Off	Use Rx FullCAN message objects (deactivate to reduce ROM and runtime consumption)

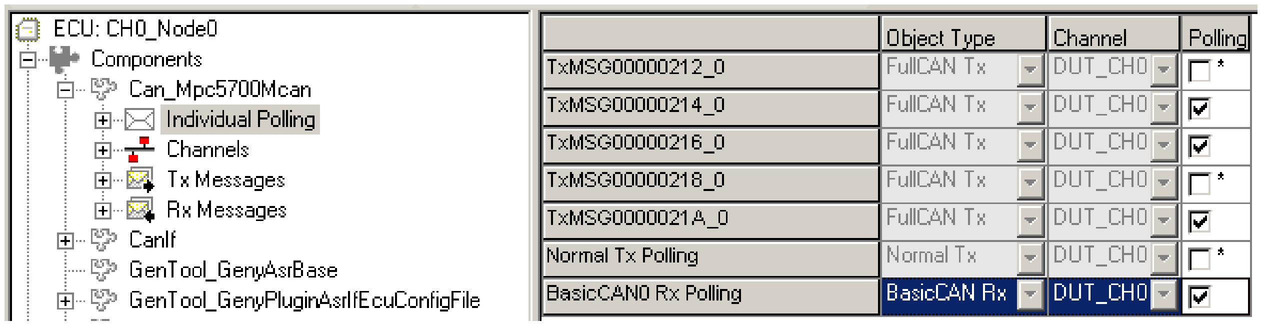
Attribute Name	Configuration Variant	Value Type	Values	Description
Tx FullCAN Support	pre-compile	Bool	On	Use Tx FullCAN message objects (deactivate to reduce ROM and runtime consumption)
Rx BasicCAN Support	pre-compile	Bool	On	Use Rx BasicCAN message objects (deactivate to reduce ROM and runtime consumption)
Use Nested CAN Interrupts	pre-compile	Bool	On / Off	Use Nested CAN interrupts (deactivate to reduce runtime consumption)
Hardware Loop Check by Application	pre-compile	Bool	On / Off	Enable this when you like to handle hardware cancellation by your own. Using API functions “ApplCanTimerStart()“, „ApplCanTimerEnd()“ and „ApplCanTimerLoop()“
Report CAN_E_TIMEOUT as DET 0x07 notification	pre-compile	Bool	On / Off	Reports DEM Timeout as DET 0x07 notification
Individual Processing	pre-compile	Bool	On / Off	Enable this if you like to configure mailbox specific polling/interrupt processing. This feature is only available with “High End” license.
Support Mixed ID	pre-compile	Bool	On / Off	Use standard and extended ID mode for Rx and Tx messages.
Type of Interrupt Function	link-time	Enum	Category 1 Category 2	Category 1: Interrupt Function has to be added to the interrupt vector table. Category 2: Interrupt Function is defined with ISR() define.
Get Status API	pre-compile	Bool	On / Off	Support Can_GetStatus() API that return the hardware states: CAN_STATUS_STOP, CAN_STATUS_INIT, CAN_STATUS_WARNING, CAN_STATUS_PASSIVE, CAN_STATUS_BUSOFF, CAN_STATUS_SLEEP

Attribute Name	Configuration Variant	Value Type	Values	Description
CAN Interrupt lock	pre-compile	Enum	CANDriver Application Both	Disable and Restore Can Interrupt by CAN Driver or Application or CAN Driver and Application. Reason may be, that the CAN Driver should not access the Interrupt controller for CAN interrupts or there are other restrictions (special security level) or Application has to care about additional Interrupts like Wakeup.
Multiple BasicCAN Objects	pre-compile	Bool	On / Off	Support Multiple BasicCAN objects. This gives the possibility to use multiple Filters and optimize acceptance Filtering as well as avoid overrun. This feature is only available with “High End” license.
Optimize for one controller	pre-compile	Bool	On / Off	If only one CAN controller is used, then the code can be optimized for space and runtime.
Dynamic FullCAN Tx ID	pre-compile	Bool	Off	Use dynamic FullCAN Tx IDs during runtime.
Size of Hw HandleType	link-time	Integer	Size	Adapt the type of hw handles for compiler options.
Partial Networking (may not be visible, depends on your delivery)	pre-compile	Bool	Off	Activate this when you like to support a partial network that will be woken up by a special NM message. This feature has to be enabled if one of the controllers shall wake up by a partial network transceiver.
RAM Check	pre-compile	Enum	None/ Active/ Mailbox Notification	RAM check of mailbox buffers. None: No RAM check Active: Active RAM check (global notification) Mailbox Notification: Active RAM check (global + mailbox notification)
Overrun Notification	pre-compile	Enum	None, DET, Application	Action to be done in case of a Rx FIFO overrun
Generic PreTransmit	pre-compile	Bool	On / Off	A generic pre-transmit, common for all Tx messages will be called when this checkbox is enabled. Use this to change data or abort transmission right before the message will be send.

Attribute Name	Configuration Variant	Value Type	Values	Description
Rx Queue				
Rx Queue	pre-compile	Bool	On / Off	Enable this when you like to handle Rx messages in polling context. (Shorten interrupt latency time). This feature is only available with “High End” license.
Size	pre-compile	Bool	On / Off	Size of FIFO in which the Message information is buffered.
Postbuild configuration (not supported for the Beta version)				
Module Start Address (Post-build only)	post-build	Integer	Address	Destination address where data has to be flashed
Max Nr. Of Tx Objects (Post-build only)	link-time	Integer	Amount	Total amount of maximum used transmit objects over all controllers. This is used for RAM variables.
Max CanController (Post-build only)	link-time	Integer	Amount	Amount of maximum used controllers (used for RAM variables).
Configuration ID (Post-build only)	post-build	Integer	Identity	A special value to identify this configuration in the Flash.

Table 8-2 Component Parameter Description

8.4.3 Individual Polling Settings



	Object Type	Channel	Polling
TxMSG00000212_0	FullCAN Tx	DUT_CH0	☐ *
TxMSG00000214_0	FullCAN Tx	DUT_CH0	✓
TxMSG00000216_0	FullCAN Tx	DUT_CH0	✓
TxMSG00000218_0	FullCAN Tx	DUT_CH0	☐ *
TxMSG0000021A_0	FullCAN Tx	DUT_CH0	✓
Normal Tx Polling	Normal Tx	DUT_CH0	☐ *
BasicCAN0 Rx Polling	BasicCAN Rx	DUT_CH0	✓

With “Individual Polling” feature (see 8.4.2) you can select for each mailbox Rx/Tx event to be polled or handled by interrupt. In case of FullCAN mailbox the name of the message is shown. In case of BasicCAN mailbox “BasicCAN Rx (X)” or “Normal Tx” has to be selected to handle all messages related to this mailbox in Polling Mode.

“Normal Tx” is the collection of all none FullCAN Tx messages.

“BasicCAN Rx (X)” is the collection of all none FullCAN Rx messages pass the BasicCAN Filter “X”.

8.4.4 Common CAN settings

Not supported.

To use the Common CAN feature the used derivative must have at least two CAN channels and at least one CAN channel must be available.

To activate the feature the ‘Use Common CAN’ checkbox in the hardware specific configuration page of the channel must be turned ‘on’ and the ‘CAN B base address’ of the ‘second’ hardware channel must be selected.

After the feature was activated on the channel it is possible to place additional Rx FullCAN message objects to the ‘second’ – ‘B’ hardware channel:

8.4.5 Controller (Channel) Settings

Add up to three different Init Structures to setup a variety of baud rate and filter settings that can be selected during the initialization of the ECU. The structure to be activated can be selected during initialization by calling the function “Can_InitStruct()” .

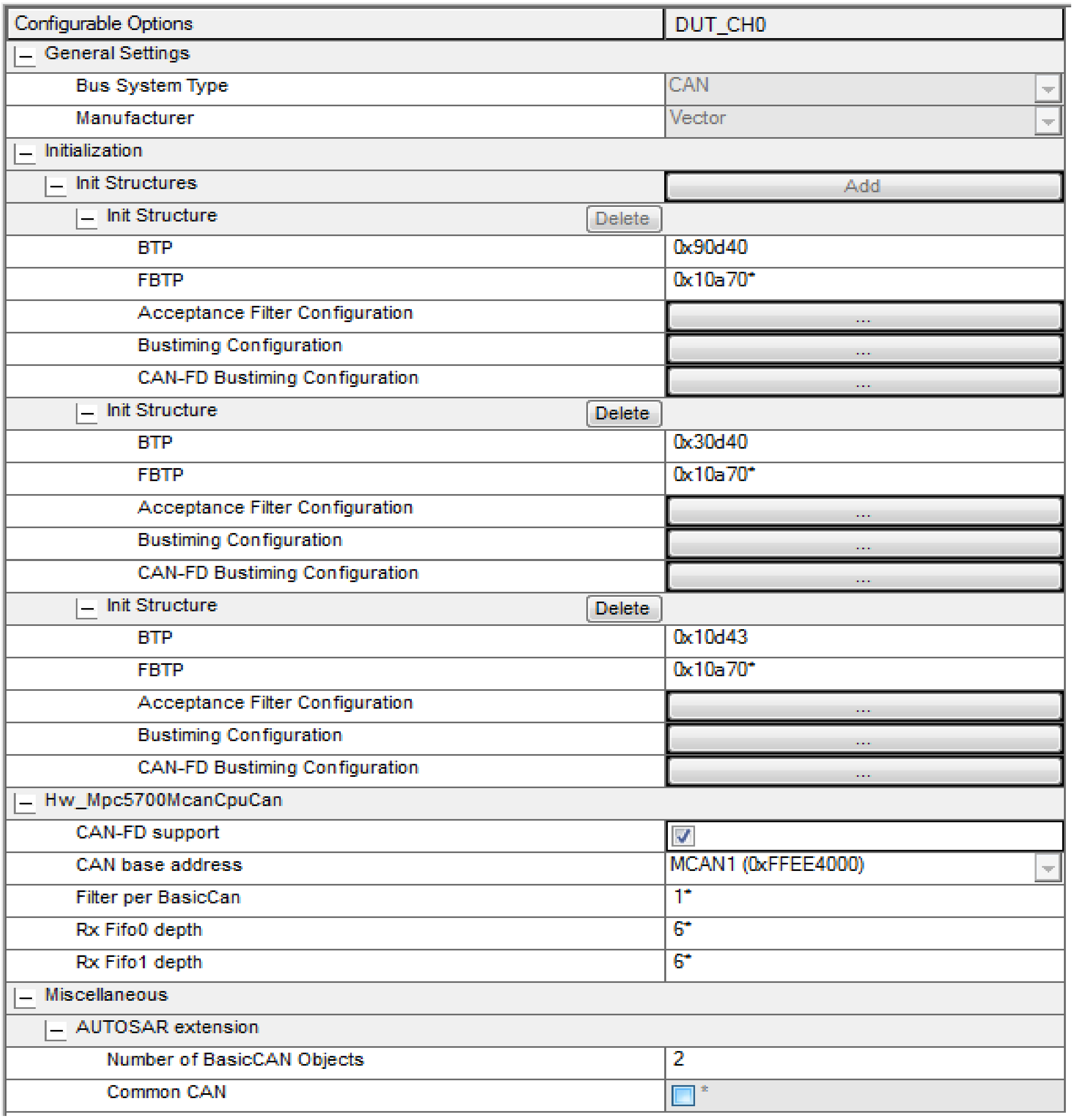
Can_InitStruct is a (None AUTOSAR API) service function that allows to change the active initialization structure left behind by the Generation Tool.

A subsequent call of Can_InitController() must be accomplished to activate these settings.

void _InitStruct(uint8 Controller, uint8 Index)

param[in]	Controller	CAN controller to be changed
param[in]	Index	Index of the initialization structure to be used

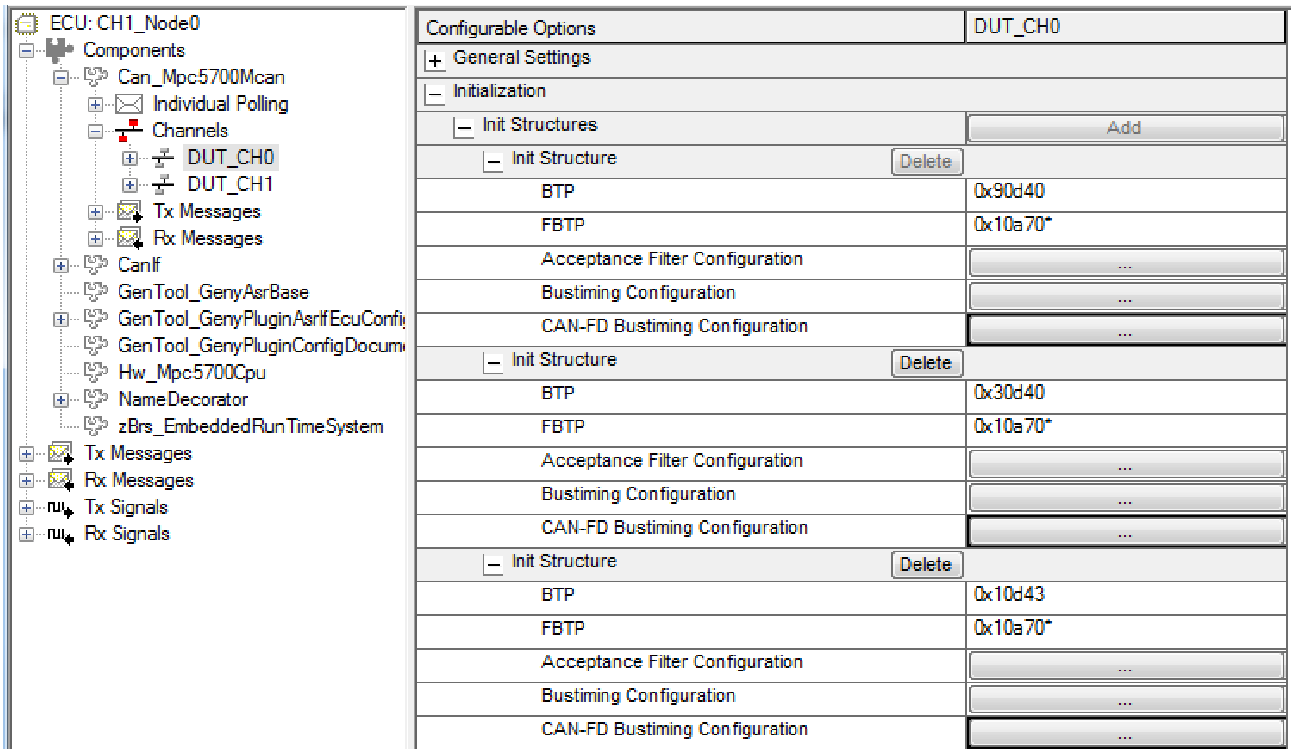
Note: Call this function between calling "Can_Init()" and Can_InitController()".



Configurable Options	DUT_CH0
General Settings	
Bus System Type	CAN
Manufacturer	Vector
Initialization	
Init Structures	Add
Init Structure Delete	
BTP	0x90d40
FBTP	0x10a70*
Acceptance Filter Configuration	...
Bustiming Configuration	...
CAN-FD Bustiming Configuration	...
Init Structure Delete	
BTP	0x30d40
FBTP	0x10a70*
Acceptance Filter Configuration	...
Bustiming Configuration	...
CAN-FD Bustiming Configuration	...
Init Structure Delete	
BTP	0x10d43
FBTP	0x10a70*
Acceptance Filter Configuration	...
Bustiming Configuration	...
CAN-FD Bustiming Configuration	...
Hw_Mpc5700McanCpuCan	
CAN-FD support	✓
CAN base address	MCAN1 (0xFFEE4000)
Filter per BasicCan	1*
Rx Fifo0 depth	6*
Rx Fifo1 depth	6*
Miscellaneous	
AUTOSAR extension	
Number of BasicCAN Objects	2
Common CAN	☐ *

Figure 8-2 Controller Settings

8.4.5.1 Init Structure Settings



ECU: CH1_Node0
- Components
  - Can_Mpc5700Mcan
    - Individual Polling
    - Channels
      - DUT_CH0
      - DUT_CH1
    - Tx Messages
    - Rx Messages
  - CanIf
  - GenTool_GenyAsrBase
  - GenTool_GenyPluginAsrIfEcuConfi
  - GenTool_GenyPluginConfigDocum
  - Hw_Mpc5700Cpu
  - NameDecorator
  - zBrs_EmbeddedRunTimeSystem
- Tx Messages
- Rx Messages
- Tx Signals
- Rx Signals

Configurable Options	DUT_CH0
General Settings	
Initialization	
Init Structures	Add
Init Structure (Delete)	
BTP	0x90d40
FBTP	0x10a70*
Acceptance Filter Configuration	...
Bustiming Configuration	...
CAN-FD Bustiming Configuration	...
Init Structure (Delete)	
BTP	0x30d40
FBTP	0x10a70*
Acceptance Filter Configuration	...
Bustiming Configuration	...
CAN-FD Bustiming Configuration	...
Init Structure (Delete)	
BTP	0x10d43
FBTP	0x10a70*
Acceptance Filter Configuration	...
Bustiming Configuration	...
CAN-FD Bustiming Configuration	...

Figure 8-3 Init Structure Dialog

Attribute Name	Configuration Variant	Value Type	Values	Description
Initialization				
BTP	link-time, post-build	Integer	"Bit Timing and Prescaler"	This value results from the "Bustiming configuration" (see below) or can alternatively be entered directly.
FBTP (optional)	link-time, post-build	Integer	"Fast Bit Timing, Prescaler and Transceiver Delay Compensation"	This value results from the "CAN-FD Bustiming configuration" (see below) or can alternatively be entered directly.
Acceptance Filter Configuration	link-time, post-build	Integer	See chapter "Acceptance Filter Settings "	Acceptance filter configuration for BasicCAN Rx objects.
Bustiming Configuration	link-time, post-build	Integer	See chapter "Bus Timing Settings"	Bit timing configuration

Attribute Name	Configuration Variant	Value Type	Values	Description
CAN-FD Bustiming Configuration	link-time, post-build	Integer	See chapter "Bus Timing Settings"	Fast Bit timing configuration

Figure 8-4 Init Structure description

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

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

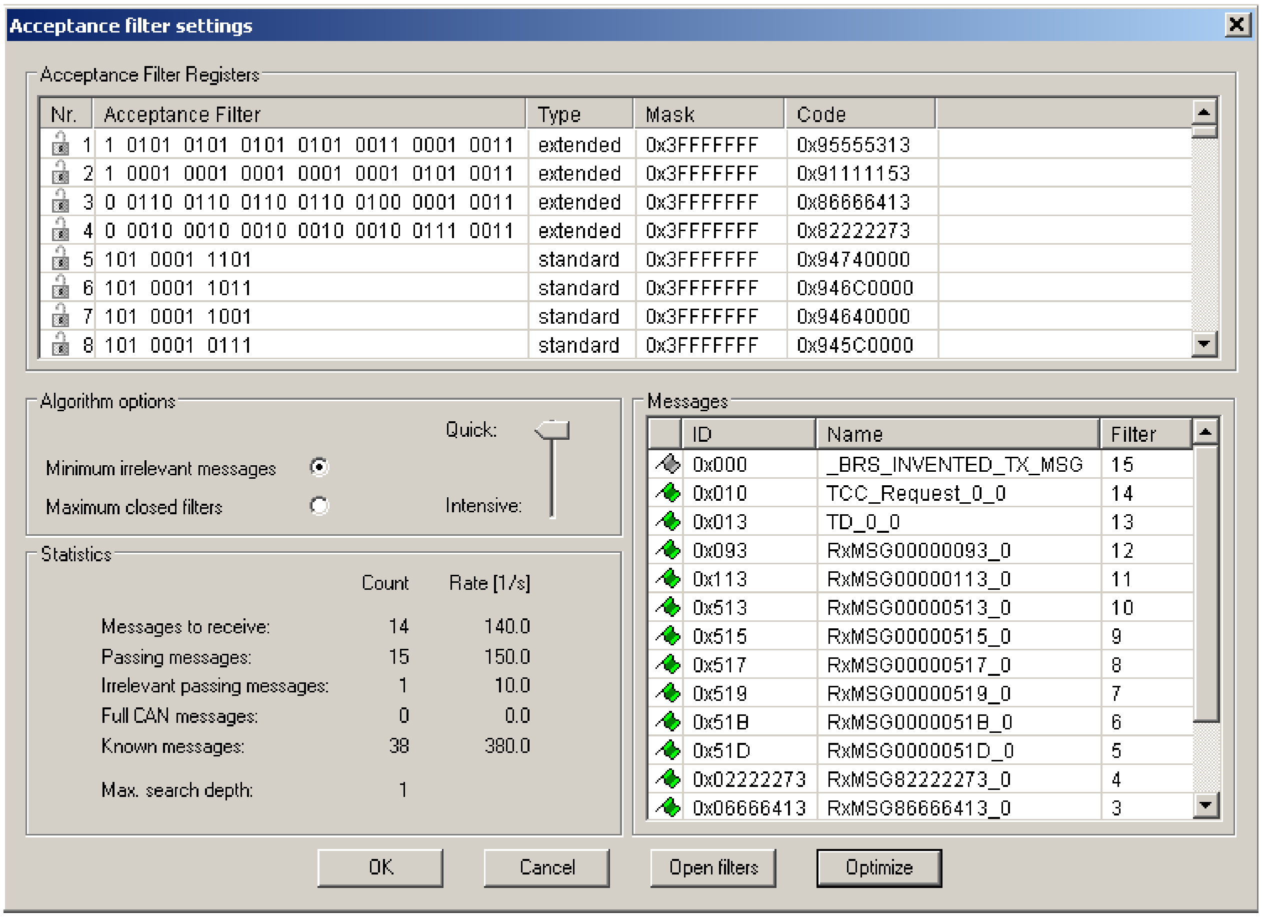
8.4.5.2 MCAN Hardware Configuration

Attribute Name	Configuration Variant	Value Type	Values	Description
Hw_Mpc5700MCanCpuCan				
CAN-FD support	pre-compile, link-time	Bool	On / Off	Activate this when you like to support CAN-FD (with bitrates higher than 1 Mbit for a fast data transfer).
MCAN BaseAddress	pre-compile, link-time	Enum	0xF7ED4000 … 0xFBEF0000	Select the base address of the CAN controller which is used on the channel.
Filters per Basic CAN	pre-compile, link-time	Integer	0 … 128 for Standard, 0 … 64 for Extended, 0 … 192 for Mixed ID	Enter the number of available filters on this channel.
Rx Fifo(0/1) depth	pre-compile, link-time	Integer	1 … 64 per Fifo	Enter the number of messages that can be stored within Fifo-0 (and Fifo-1 in case of MultipleBasicCAN).

Figure 8-5 Hardware description

8.4.5.3 Acceptance Filter Configuration

Setup the Filter by pressing the “Acceptance Filter Configuration” button.



Acceptance filter settings

Acceptance Filter Registers

Nr.	Acceptance Filter	Type	Mask	Code
1	1 0101 0101 0101 0101 0011 0001 0011	extended	0x3FFFFFFF	0x95555313
2	1 0001 0001 0001 0001 0001 0101 0011	extended	0x3FFFFFFF	0x91111153
3	0 0110 0110 0110 0110 0100 0001 0011	extended	0x3FFFFFFF	0x86666413
4	0 0010 0010 0010 0010 0010 0111 0011	extended	0x3FFFFFFF	0x82222273
5	101 0001 1101	standard	0x3FFFFFFF	0x94740000
6	101 0001 1011	standard	0x3FFFFFFF	0x946C0000
7	101 0001 1001	standard	0x3FFFFFFF	0x94640000
8	101 0001 0111	standard	0x3FFFFFFF	0x945C0000

Algorithm options

Minimum irrelevant messages [✓]

Maximum closed filters

Quick:

Intensive:

Statistics

	Count	Rate [1/s]
Messages to receive:	14	140.0
Passing messages:	15	150.0
Irrelevant passing messages:	1	10.0
Full CAN messages:	0	0.0
Known messages:	38	380.0
Max. search depth:	1	

Messages

ID	Name	Filter
0x000	_BRS_INVENTED_TX_MSG	15
0x010	TCC_Request_0_0	14
0x013	TD_0_0	13
0x093	RxMSG00000093_0	12
0x113	RxMSG00000113_0	11
0x513	RxMSG00000513_0	10
0x515	RxMSG00000515_0	9
0x517	RxMSG00000517_0	8
0x519	RxMSG00000519_0	7
0x51B	RxMSG0000051B_0	6
0x51D	RxMSG0000051D_0	5
0x02222273	RxMSG82222273_0	4
0x06666413	RxMSG86666413_0	3

Figure 8-6 Setup Filter Dialog

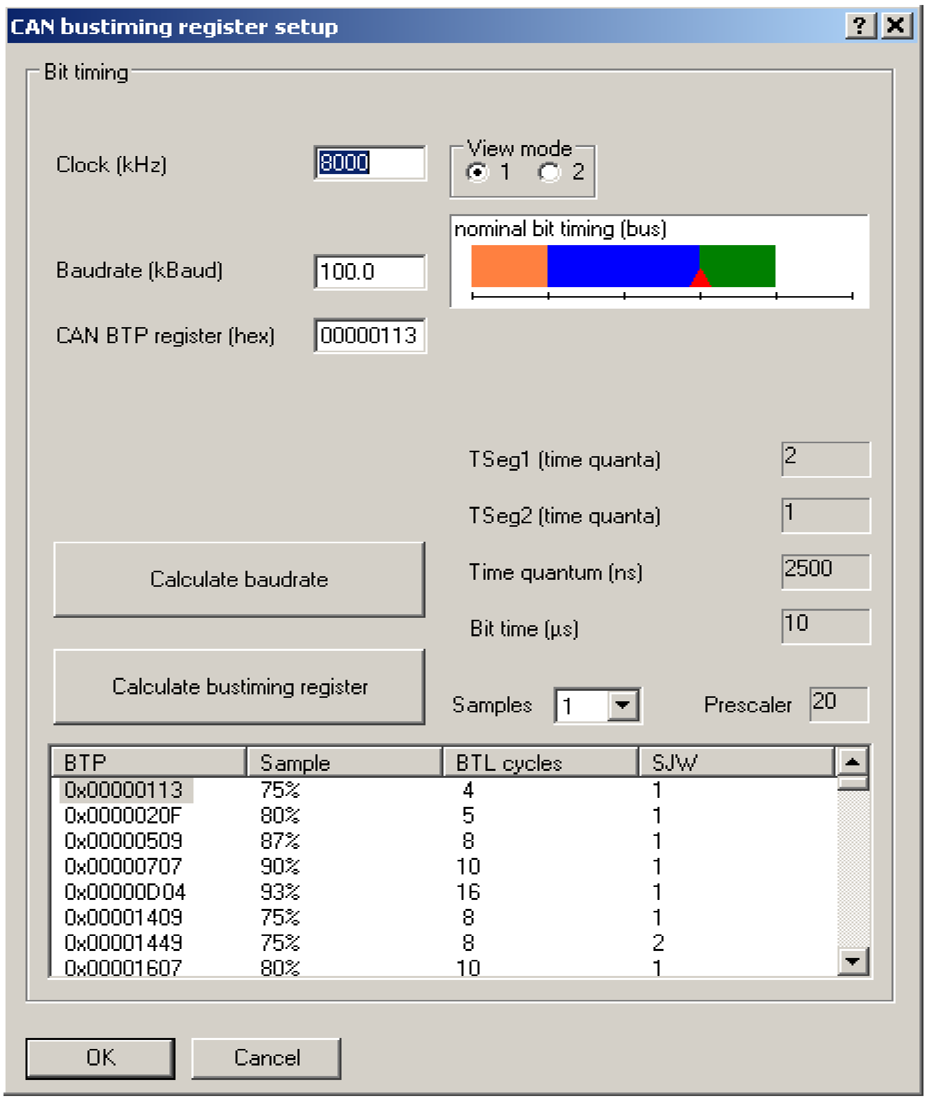
Attribute Name	Configuration Variant	Value Type	Values	Description
Setup				
Acceptance Filter	post-build	Integer	Mask and Code register	Each ID –bit is represented by a “0/1/X”, means must match a “0” or “1” or must not match “X”
Open filters		Action	-	Open the filter completely to receive all IDs
Optimize		Action	-	Open the filter automatic to just receive the IDs in database. “Use FullCAN” try to put as much messages in FullCAN objects to better optimize filters

Table 8-1 Filter Parameter Description

Caution
The hardware supports a maximum of 128 standard and 64 extended filters for each CAN channel. For each Rx FullCAN object one filter is grabbed by the generation tool. Your configuration must not exceed the total available number of standard/extended filters used for FullCAN and BasicCAN objects.

8.4.5.4 Bus Timing Configuration

Setup the Baud rate by pressing the “Bustiming Configuration” button.



Clock (kHz): 8000
View mode: [✓ 1] 2
Baudrate (kBaud): 100.0
CAN BTP register (hex): 00000113
TSeg1 (time quanta): 2
TSeg2 (time quanta): 1
Time quantum (ns): 2500
Bit time (µs): 10
Samples: 1
Prescaler: 20

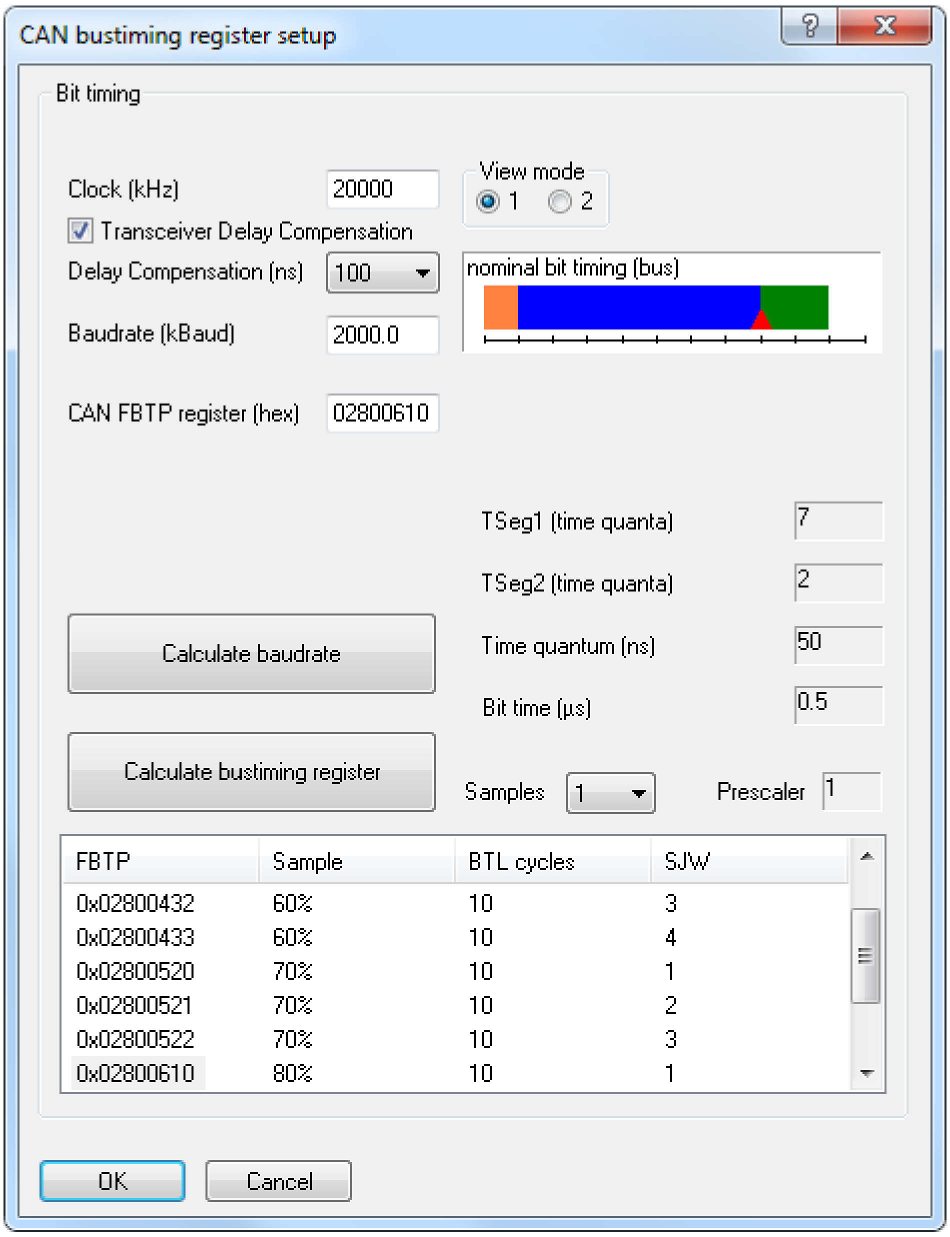
BTP	Sample	BTL cycles	SJW
0x00000113	75%	4	1
0x0000020F	80%	5	1
0x00000509	87%	8	1
0x00000707	90%	10	1
0x00000D04	93%	16	1
0x00001409	75%	8	1
0x00001449	75%	8	2
0x00001607	80%	10	1

Figure 8-7 Baud Rate Dialog

Attribute Name	Configuration Variant	Value Type	Values	Description
Clock	post-build	Integer	CAN clock	Set the clock of the CAN engine, i.e. oscillator clock or system clock.
Baud rate	post-build	Integer	Baud Rate	Set baud rate to be used for this channel
Calculate		Action	-	It is possible to calculate possible hardware register settings out of baud rate or vice versa.
Sample, BTL cycles, SJW	post-build	Select Set of Values	Baud Rate Settings	Select the sample point and sync phase related to your bus physics

Table 8-2 Baud Rate Parameter Description

Setup the CAN-FD Baud rate by pressing the “CAN-FD Bustiming Configuration” button.



CAN bustiming register setup

Bit timing

Clock (kHz): 20000

View mode: [✓ 1] 2

[✓] Transceiver Delay Compensation

Delay Compensation (ns): 100

nominal bit timing (bus)

Baudrate (kBaud): 2000.0

CAN FBTP register (hex): 02800610

TSeg1 (time quanta): 7

TSeg2 (time quanta): 2

Time quantum (ns): 50

Bit time (µs): 0.5

Samples: 1

Prescaler: 1

FBTP	Sample	BTL cycles	SJW
0x02800432	60%	10	3
0x02800433	60%	10	4
0x02800520	70%	10	1
0x02800521	70%	10	2
0x02800522	70%	10	3
0x02800610	80%	10	1

Figure 8-8 CAN-FD Baud Rate Dialog

Attribute Name	Configuration Variant	Value Type	Values	Description
Clock	post-build	Integer	CAN clock	Set the clock of the CAN engine, i.e. oscillator clock or system clock.
Transceiver Delay Compensation	post-build	Bool	On / Off	If selected the transmitter measures within each CAN FD frame the delay between the data transmitted at pin M_CAN_Tx and the data received at M_CAN_Rx pin. A secondary sample point is calculated by adding the configurable transceiver delay compensation (see below) to the measured transceiver delay. The delay is measured in M_CAN core clock periods.
Delay Compensation	post-build	Integer	CAN-FD Delay Compensation	A secondary sample point is calculated by adding this configurable transceiver delay compensation to the measured transceiver delay (see above). The transceiver delay compensation is used to adjust the secondary sample point inside the bit time. The position of the secondary sample point is rounded down to the next integer number of the time quanta. To check for bit errors during the data phase, the delayed transmit data is compared against the received data at the secondary sample point. If a bit error is detected at the secondary sample point, the transmitter will react to this bit error at the next following sample point.
Baud rate	post-build	Integer	CAN-FD Baud Rate	Set baud rate to be used for this channel
Calculate		Action	-	It is possible to calculate possible hardware register settings out of baud rate or vice versa.
Sample, BTL cycles, SJW	post-build	Select Set of Values	Baud Rate Settings	Select the sample point and sync phase related to your bus physics

Table 8-3 CAN-FD Baud Rate Parameter Description

Note
For the transceiver delay compensation the following boundary conditions have to be considered:

- The sum of the measured delay from M_CAN_Tx to M_CAN_Rx and the configured transceiver delay compensation has to be less than 3 bit times in the data phase.
- The sum of the measured delay from M_CAN_Tx to M_CAN_Rx and the configured transceiver delay compensation has to be less or equal 63 M_CAN core clock periods. In case this sum exceeds 63 M_CAN core clock periods, the maximum value of 63 M_CAN core clock periods is used for transceiver delay compensation.
- The actual delay compensation value is monitored by reading the MCAN test register TEST(TDCV).

8.5 Configuration with da DaVinci Configurator

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

9 AUTOSAR Standard Compliance

9.1 Limitations / Restrictions

Category	Description	Version
Functional	No multiple AUTOSAR CAN driver allowed in the system	3.0.6
Functional	No support for L-PDU callout (AUTOSAR 3.2.1), but support 'Generic Precopy' instead	3.2.1
Functional	No support for multiple read and write period configuration	3.2.1
API	"Symbolic Name Values" may change their values after precompile phase so do not use it for Link Time or Post Build variants. It's recommended that higher layer generator use Values (ObjectIDs) from EcuC file. Vector CAN Interface does so.	3.0.6
	For the acceptance filtering a maximum of 64 filters per CAN channel is supported in case of GENy is used as Generation Tool.	

9.2 Hardware Limitations

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 behaviour.

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 (see below on how to) and use your own initialization instead of.

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

```
#define CAN_ECC_INIT     STD_OFF
```

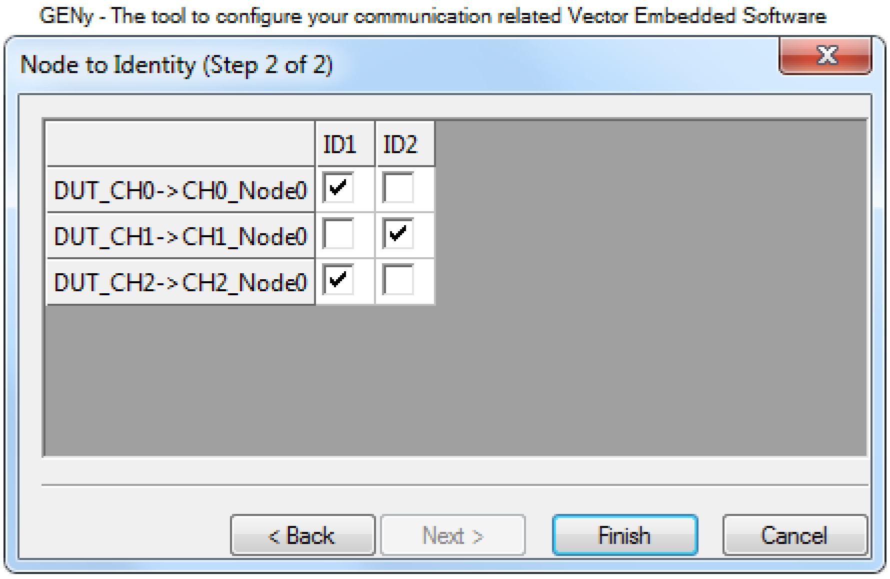
Put your initialization into execution just before calling Can_Init(). The MCAN clock must be available at this point of time.

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

9.2.5 Multiple configuration

When using the Multiple Configurations feature (only available with GENy as Generation Tool) you must care of the physical CAN channel assignment to the logical CAN channels. Due to the common CAN Message RAM usage of several CAN channels it has to be avoided that the allocated Message RAM partitions per CAN channel overlap each other.

Example (1): Correct configuration with 3 logical channels
using 2 physical channels within 2 identities



GENy - The tool to configure your communication related Vector Embedded Software

Node to Identity (Step 2 of 2)

	ID1	ID2
DUT_CH0->CH0_Node0	✓	
DUT_CH1->CH1_Node0		✓
DUT_CH2->CH2_Node0	✓	

Figure 9-1 Correct Identity Mapping example

Let us assume that the logical channels “DUT_CH0” and “DUT_CH1” use the same physical channel “M_CAN_2” and “DUT_CH2” is mapped on the physical channel “M_CAN_4”.
The CAN Message RAM is allocated per logical channel starting with the first one.

If the first logical channel “DUT_CH0” (see Figure 9-1) has a different configuration (allocating more or less CAN Message RAM) than the subsequent logical channel “DUT_CH1”, both mapped to the first physical channel “M_CAN_2”, then the subsequent channels within one and the same identity must assure that there is no address conflict (overlap or gap) within the CAN Message RAM allocation.

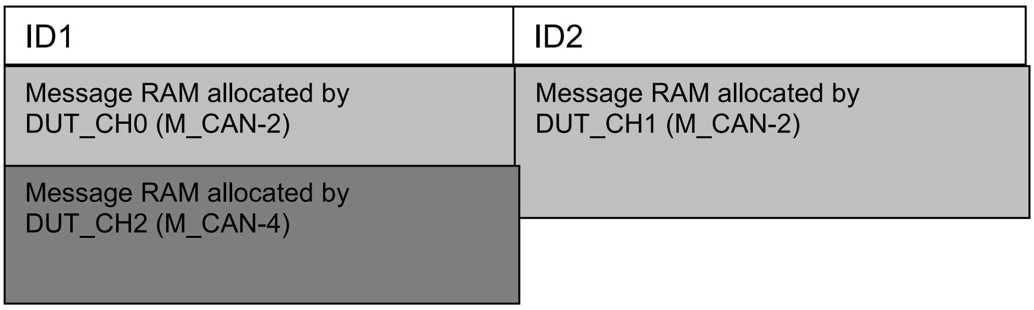
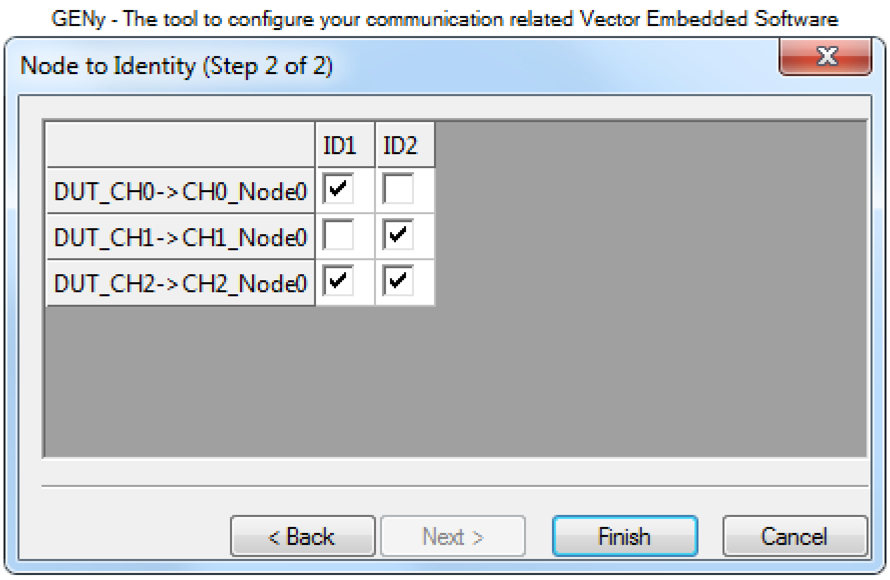

ID1
ID2
Message RAM allocated by DUT_CH0 (M_CAN-2)
Message RAM allocated by DUT_CH1 (M_CAN-2)
Message RAM allocated by DUT_CH2 (M_CAN-4)

Figure 9-2 Correct CAN Message RAM allocation example

Example (2): Faulty configuration with 3 logical channels
using 2 physical channels within 2 identities

If you modify the identity mapping in the way that the previous channel allocation is changed like shown in “Figure 9-3 Faulty Identity Mapping” then you must assure that no CAN Message RAM overlap can appear (see “Figure 9-4 Faulty CAN Message RAM allocation example”Figure 9-4).

In the example below the “M_CAN_2” configuration is partly overwritten by the “M_CAN_4” configuration when using Identity-2:


GENy - The tool to configure your communication related Vector Embedded Software
Node to Identity (Step 2 of 2)
< Back
Next >
Finish
Cancel

	ID1	ID2
DUT_CH0->CH0_Node0	✓	
DUT_CH1->CH1_Node0		✓
DUT_CH2->CH2_Node0	✓	✓

Figure 9-3 Faulty Identity Mapping example

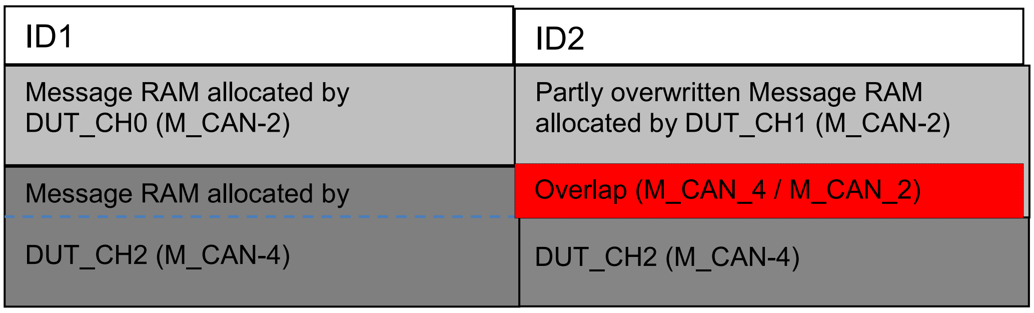

ID1
Message RAM allocated by DUT_CH0 (M_CAN-2)
Message RAM allocated by
DUT_CH2 (M_CAN-4)
ID2
Partly overwritten Message RAM allocated by DUT_CH1 (M_CAN-2)
Overlap (M_CAN_4 / M_CAN_2)
DUT_CH2 (M_CAN-4)

Figure 9-4 Faulty CAN Message RAM allocation example

Note

(The Message RAM Start/End addresses of each channel can be checked within the generated table “CanShmStartEndAdr”.)

9.3 Vector Extensions

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

10 Glossary and Abbreviations

10.1 Glossary

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

Table 10-1 Glossary

10.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
DEM	Diagnostic Event Manager
DET	Development Error Tracer
ECU	Electronic Control Unit
HIS	Hersteller Initiative Software
ISR	Interrupt Service Routine
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution) 3.3x = AUTOSAR version 3 401 = AUTOSAR version 4.0.1 403 = AUTOSAR version 4.0.3 4x = AUTOSAR version 4.x.x
SWS	Software Specification
Common CAN	Connect two physical peripheral channels to one CAN bus (to increase the amount of FullCAN 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