VECTOR

MICROSAR Classic CAN Interface

Technical Reference

Version 8.11.00

Author	visgaz
Status	Released

Document Information

History

Author	Date	Version	Remarks
visseu, visrna	2012-07-17	5.00	ASR R4.0 Rev 3
visseu	2013-04-03	5.01.00	ESCAN00065368 ESCAN00066338 ESCAN00066340 Adapted according to ESCAN00066285 Adapted according to ESCAN00065289 ESCAN00066396 Adapted according to ESCAN00064304
visrna	2013-07-24	5.01.01	ESCAN00066794
visseu	2013-09-27	6.00.00	Adapted due to: AR4-307: J1939 support AR4-438: Dynamic address lookup table AR4-397: CAN FD support
visseu	2014-05-19	6.01.00	CAN FD support extended: Rx-FD and Rx- and Tx-PDUs with up to 64 bytes payload
visrna	2014-07-10	6.02.00	Multiple CAN Driver support
visseu	2014-08-25	6.02.00	ESCAN00077304, Restriction concerning the handling of FD/Not-FD FullCAN-Rx-PDUs added
visseu	2014-09-22	6.02.00	ESCAN00078524, CanTSyn added, Post-build selectable
visseu	2014-11-25	6.03.00	Channel specific J1939 dynamic address
visseu	2015-01-26	6.04.00	Chapter 2.8 adapted to changed implementation
visseu	2015-05-18	6.05.00	Adapted due to FEAT-366
visseu	2015-11-20	6.06.00	Adapted due to FEAT-1429
visseu	2016-01-09	6.06.00	ESCAN00087340
visseu	2016-02-22	6.07.00	Feature Extended RAM-check added, ESCAN00087587
visseu	2016-06-24	6.08.00	Feature: Data checksum added
visseu	2016-09-14	6.09.00	Adapted due to FEAT-2076: Behavior of Tx-PDU filter extended
visseu	2016-09-26	6.09.00	Adapted due to FEAT-2024: Set reception mode
visseu	2017-01-09	6.10.00	Improved due to ESCAN00093454
visseu	2017-02-13		Adapted due to: FEAT-2140: TMC Checksum - Release feature FEAT-1914
visseu	2017-02-28		ESCAN00094196, deviation from AUTOSAR documented by ESCAN00094121 added
visseu	2017-08-04	6.11.00	ESCAN00096181

Author	Date	Version	Remarks
visseu	2017-08-30	6.11.01	Typos corrected
visgaz	2018-07-11	6.12.00	Added API description for CanIf_CheckBaudrate() (ESCAN00094506), extended Service IDs in Table 2-7, extended error codes in Table 2-8, removed CanIf_InitController() in chapter 2.3.1, removed CanIf_Hooks.h in Table 3-1
visgaz	2019-02-04	6.13.00	Removed obsolete Service ID in Table 2-7, replaced CAN channel with CAN controller in Table 2-8, reworked Chapter 5, added BusMirroring feature
visgaz	2019-02-27	6.14.00	Adapted deviation 7.2.5 Check wakeup, added Tx-PDU truncation feature
visgaz	2019-06-12	7.00.00	Reworked Feature List and document structure, added API descriptions for CanIf_ClearTrcvWufFlag() / CanIf_CheckTrcvWakeFlag() (ESCAN00103412),
visgaz	2020-01-22	7.01.00	Reworked Chapter 3.2 - Critical Sections (ESCAN00104612), Reworked API descriptions
visgaz	2020-06-04	7.02.00	Added possible blocking situation description for Tx-Buffer of type FIFO to Chapter 2.5.3.1
visgaz	2020-08-04	7.03.00	Added security event reporting feature, made formal adaptions
visgaz	2020-08-31	7.04.00	Added error code `CANIF_E_TX_BUFFER_TRANSMIT`
visgaz	2020-11-23	7.05.00	Adapted security event reporting feature according to AR20-11 (CAN-1989)
visgaz	2021-01-04	7.06.00	Added priority description for BasicCAN Rx-PDUs (CAN-1972)
visgaz	2021-01-18	8.00.00	Made formal adaptions (CAN-2032), Added new used VStdLib APIs to Table 4-35 (CAN-36)
visgaz	2021-02-22	8.00.01	Adapted CanIf_MemMap.h description in Table 3-1 (CAN-2567)
visgaz	2021-03-11	8.01.00	Added support ASR4.4.0 CAN Driver feature (CAN-2553)
visgaz	2021-05-05	8.02.00	Removed support of configuration variant Link-Time (CAN-2592), Adapted CanIf_MemMap.h description (CAN-2588)
visgaz	2022-02-25	8.03.00	Added multi-partition support (CANCORE-463), renamed MICROSAR to MICROSAR Classic

Author	Date	Version	Remarks
visgaz	2022-03-25	8.03.01	Improved multi-partition support description (CANCORE-1052) and MICROSAR Classic wording
visgaz	2022-09-01	8.04.00	Added support of two CAN Drivers with different AUTOSAR versions (CANCORE-1373)
destler	2023-02-22	8.05.00	Remove configuration options for Tx-Buffer and Software filtering (CANCORE-1199)
visgaz	2023-03-27	8.06.00	Added memory mapping description for CANIF_APPL_VAR (CANCORE-1387) Reworked description for CAN Drivers with different AUTOSAR versions (CANCORE-1833) Adapted configuration option description for software filtering
visgaz	2023-04-28	8.07.00	Reworked description for supported CAN Drivers (CANCORE-1835) Improved description for CanIf_SetControllerMode and exclusiv areas (CANCORE-493)
visgaz	2023-06-26	8.08.00	Added support of dynamic Rx PDUs (CANCORE-1771) Improved description for J1939 dynamic address support (CANCORE-2012)
destler	2023-08-04	8.09.00	Added support for CAN Drivers of AR22-11 (CANCORE-2106) Added API descriptions for CanIf_GetTxAddressTableEntry and CanIf_GetRxAddressTableEntry (CANCORE-2096)
visgaz, destler	2023-11-21	8.10.00	Added CAN XL (CANCORE-2088, CANCORE-1706, CANCORE-2231) Improved API description for CanIf_CheckValidation() Added software filter type DOUBLEHASH to Table 2-1
visgaz	2024-03-13	8.11.00	Removed CAN transceiver Drivers restriction for multi-partition in chapter 2.21 (CANCORE-2527)

Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	Specification of CAN Interface	R4.2.2
[2]	AUTOSAR	Specification of Default Error Tracer	R4.2.2
[3]	AUTOSAR	List of Basic Software Modules	R4.2.1
[4]	AUTOSAR	Specification of CAN Interface	R22-11

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

Contents

Illustrations

Tables

1 Introduction

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

Supported Configuration Variants:	Pre-compile, Post-build	
Vendor ID:	CANIF_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	CANIF_MODULE_ID	60 decimal (according to ref. [3])

The CAN Interface is a hardware independent layer with a standardized interface to the CAN Driver and CAN Transceiver Driver layer and upper layers like PDU Router, Communication Manager and the Network Management. All upper layers which require CAN communication have to communicate with the CAN Interface which is responsible for the CAN communication handling. This includes the transmission and the reception of messages and the state handling of the CAN controllers as well.

1.1 Architecture Overview

The next figure shows the interfaces to adjacent modules of the CAN Interface. These interfaces are described in chapter 4.

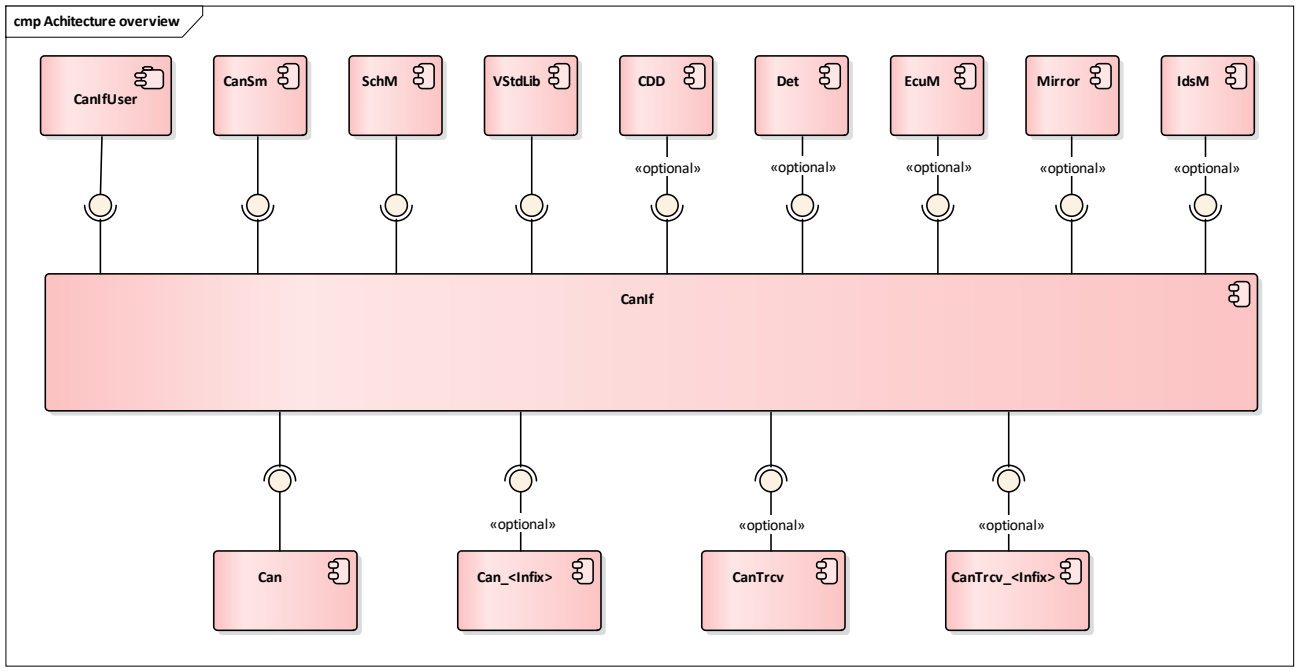

Figure 1-1 Interfaces to adjacent modules of the CAN Interface

The modules which are informed by the CAN Interface about a successfully transmission or reception of PDUs are abstracted by the module `CanIfUser`. These modules are listed in Table 4-35 with the APIs `<Msn>_RxIndication`, `User_RxIndication`, `<Msn>_TxConfirmation` or `User_TxConfirmation`.

2 Functional Description

2.1 Features

The features listed in the following tables cover the functionality specified for the CAN Interface.

The AUTOSAR standard functionality is specified in [1], the corresponding features are listed in the tables

> Table 2-1 Supported AUTOSAR standard conform features

> Table 2-2 Not supported AUTOSAR standard conform features

Vector Informatik provides further CAN Interface functionality beyond the AUTOSAR standard. The corresponding features are listed in the table

> Table 2-3 Features provided beyond the AUTOSAR standard

The following features specified in [1] are supported:

Supported AUTOSAR Standard Conform Features
CAN Interface initialization
Transmission incl. Transmit confirmation
Transmit buffer (Prioritization by CAN-identifier) *Buffering (send request and data) of CAN 2.0/FD Tx PDUs mapped to a Tx buffer in the CAN Interface* prioritized by CAN-identifier.
Reception incl. Receive indication
DLC check *Check DLC of received Rx-PDUs against predefined values.*
CAN FD support
Meta data Rx- / Tx-support *Support for dynamic CAN identifier handling by using of SDU meta data.*
Communication Modes *Modes for CAN controllers and PDUs.*
BusOff detection *Handling of bus off notifications.*
External wakeup (CAN) *Support external wakeup by CAN Driver.*
External wakeup (Transceiver) *Support external wakeup by CAN Transceiver Driver.*
Wakeup validation *Support wakeup validation for external wakeup events.*
Error notification with Default Error Tracer (DET)
Tx Basic-CAN / Tx Full-CAN / Rx Basic-CAN / Rx Full-CAN *Basic: Standard CanHardwareObject to send/ receive CAN 2.0/FD PDUs.* *Full: Separate CanHardwareObject for special CAN 2.0/FD PDUs used.*

Supported AUTOSAR Standard Conform Features
CAN transceiver handling *APIs for upper layers to set and read transceiver states; Interface to the CAN Transceiver Driver.*
Version API *API to read out component version.*
Supported ID types (Standard Identifiers, Extended Identifiers, Mixed Identifiers) *Support of CAN Standard (11 bits) identifiers or CAN Extended (29 bits) identifiers or both.*
Multiple CAN networks *Each CAN network has to be connected to exactly one controller.*
Multiple CAN Driver support
Tx Confirmation Polling Support *This service provides the information on whether any Tx confirmation has occurred for a CAN controller since the last start of that CAN controller at all.*
Post-build loadable *Post-build loadable allows the re-configuration of an ECU at Post-build time.*

Table 2-1 Supported AUTOSAR standard conform features

2.1.1 Deviations

The following features specified in [1] are not supported or supported with deviations:

Category	Description
Functional	Dynamic transmit L-PDU handles *The priority of a dynamic Tx-PDU is determined from the initial configured CAN identifier and not from the CAN identifier set by using the API* `CanIf_SetDynamicTxId()`
Functional	Partial Networking *The Partial Networking Wakeup Tx-PDU Filter is enabled only if the PDU mode of CAN Interface is set either to mode* `CANIF_GET_TX_ONLINE_WU_FILTER` *or to mode* `CANIF_GET_ONLINE_WU_FILTER`*.*
Functional	Version check *The CAN Interface does not perform AUTOSAR release version check in accordance with other modules because the version check is not specified by AUTOSAR clearly.*
Functional	Read Tx/Rx notification status
Functional	Read received data
Functional	Trigger transmit
Functional	Pretended Networking
Config	CanIfBufferSize *Only one PDU instance of each CAN 2.0/FD Tx PDU which is assigned to a Transmit buffer (Prioritization by CAN-identifier) can be buffered. The parameter is set automatically by DaVinci Configurator Classic.*
Config	CanIfPrivateSoftwareFilterType *Only the software filter types LINEAR and DOUBLEHASH (MICROSAR Classic extension) are supported.*

Category	Description
Config	CanIfRxPduDlc *A maximum value of 2048 is configureable instead of the AUTOSAR specified maximum value of 64.*

Table 2-2 Not supported AUTOSAR standard conform features

2.1.2 Additions/Extensions

The following features are provided beyond the AUTOSAR standard:

Features Provided Beyond The AUTOSAR Standard
Software filter type DOUBLEHASH *Search algorithm for Basic-CAN CAN 2.0/FD Rx PDUs with single CAN ID.*
Hardware transmit cancellation *Avoidance of inner priority inversion (described in chapter 2.5.3.2).*
Software transmit cancellation *CanIf_CancelTransmit() contains the functionality to cancel an requested Tx-PDU and suppress its confirmation to the upper layer.*
Transmit-buffer handling type FIFO *Buffering (send request and data) of CAN 2.0/FD Tx PDUs mapped to a Tx buffer in the CAN Interface* in the requested order.
Multiple transmit-buffers per CAN controller *Per CAN controller multiple independent transmit-buffers may be configured with different handling types: FIFO or prioritization by CAN identifier.*
Control of reception mode of a Rx-PDU *This feature provides the ability to control the reception mode of a Rx-PDU individually at runtime.*
J1939 Dynamic Address Support *Translating of addresses according to J1939 by using of dynamic address lookup tables which are maintained by J1939Nm.*
Data checksum Rx *Verification of checksum of Rx-PDUs.*
Data checksum Tx *Appending of checksum to Tx-PDUs.*
Internal wakeup *Internal wakeup by calling* `CanIf_SetControllerMode()`.
Extended RAM-check *This service provides the ability in order to request an underlying CAN controller to execute a check of CAN controller HW-registers. The usage of this feature requires a corresponding license.*
Post-build selectable *MICROSAR Classic identity manager using Post-build selectable.*
Tx-PDU truncation Ability to truncate the Tx-PDU length, if it exceeds the configured length (a*ccording to AUTOSAR* release version *4.4.0)*

Features Provided Beyond The AUTOSAR Standard
Bus Mirroring *Reports transmitted and received PDUs to the Bus Mirroring module (based on AUTOSAR* release version *4.4.0 with deviations as described in chapter 2.17).*
Security event reporting *Reports occurred security events at a CAN controller to the Intrusion Detection System Manager module (based on AUTOSAR* release version *20-11).*
Multi-partition *CAN controllers can be mapped to different partitions.*
CAN Drivers with different AR versions *Support of third party CAN Drivers with different AUTOSAR versions and in combination with MICROSAR Classic CAN Drivers.*
Dynamic Rx PDUs *Supports reception of CAN frames in dynamic Rx PDUs whose CAN IDs are set during initialization via a callout function.*
CAN XL *Supports transmission and reception of CAN XL PDUs (based on AUTOSAR release version 22-11).*

Table 2-3 Features provided beyond the AUTOSAR standard

2.2 Initialization

Several functions are available to initialize the CAN Interface. The following code example shows which functions have to be called to initialize the CAN Interface and to allow transmission and reception.

```
CanIf_InitMemory();
      /* Mandatory call which reinitializes global variables to
                  set the CAN Interface back to uninitialized
                  state. */


CanTrcv_xxx_InitMemory() and CanTrcv_xxx_Init()
   /* have to be called to initialize the CAN Transceiver Driver
                  and set the CAN Transceiver to the preconfigured
                  state. For some CAN Controllers it is necessary
                  to have a recessive signal on the Rx Pin to be
                  able to initialize the CAN Controller. This
                  means the CAN transceiver has to be set to
                  "normal mode" before CanIf_Init() is called. */


Can_InitMemory() and Can_Init();
   /* have to be called before CanIf_Init is called. */
```

```
CanIf_Init(<PtrToCanIfConfiguration>);
   /* Global initialization of the CAN Interface: all available CAN
                  Interface controllers are initialized within
                  this call. If postbuild-selectable configuration
                  is active a valid configuration has to be passed
                  to CanIf_Init. In other cases the parameter is
                  ignored and a NULL pointer can be used */


CanIf_SetControllerMode(0, CANIF_CS_STARTED);
   /* The controller mode of CAN controller 0 is set to started
                  mode. This means the CAN controller is
                  initialized and ready to communicate
                  (acknowledge of the CAN controller is
                  activated). Communication is not yet possible
                  because the CAN Interface will neither pass Tx
                  PDUs from higher layers to the CAN Driver nor
                  accept Rx PDUs from the CAN Driver. */


CanIf_SetPduMode(0, CANIF_SET_ONLINE);
   /* The PDU mode in the CAN Interface of the CAN controller 0 is
                  switched to online mode. After initialization
                  this mode remains in the state CANIF_GET_OFFLINE
                  until the CanIf_SetPduMode function is called.
                  Now transmission requests will be passed from
                  the upper layer to the CAN Driver and Rx PDUs
                  are forwarded from the CAN Driver to the
                  corresponding higher layer. */
```

During the CAN Interface initialization with `CanIf_Init()` the `EcuM_BswErrorHook()` is called and the initialization will be aborted when at least one of the following errors occures:

- Pointer to the configuration is invalid (`ECUM_BSWERROR_NULLPTR`)
- Generator is not compatible (`ECUM_BSWERROR_COMPATIBILITYVERSION`)
- Magic number is wrong (`ECUM_BSWERROR_MAGICNUMBER`)

2.3 Communication Modes

The CAN Interface knows two main types of communication modes.

2.3.1 Controller Mode

The controller mode represents the physical state of the CAN controller. The following modes are available:

- `CANIF_CS_STOPPED`
- `CANIF_CS_STARTED`
- `CANIF_CS_SLEEP`
- `CANIF_CS_UNINIT`

There is no mode called bus off. Bus off is treated as a transition from STARTED to STOPPED mode. All transitions have to be initiated using the API function `CanIf_SetControllerMode()`. The controller mode can be switched for each controller independent of the mode of other controllers in the system.

The mode `CANIF_CS_UNINIT` is the default mode after power on. It is left to mode `CANIF_CS_STOPPED` after `CanIf_Init()` is called and can only be entered again by a reset of the ECU.

The modes `CANIF_CS_SLEEP` and `CANIF_CS_STARTED` can only be entered from `CANIF_CS_STOPPED`. This means a transition from STARTED to SLEEP and vice versa is not possible without requesting the STOPPED mode first.

It is always possible to request the current active controller mode by calling the API `CanIf_GetControllerMode()`.

2.3.2 PDU Mode

The other type of communication mode is completely processed by software (it does not represent any state of the hardware). Transitions of the PDU mode are only possible if the controller mode is set to `CANIF_CS_STARTED`.

The following PDU modes are available:

- `CANIF_GET_OFFLINE`

  Rx and Tx path are switched offline

- `CANIF_GET_RX_ONLINE`

  Rx path online, Tx path offline

- `CANIF_GET_TX_ONLINE`

  Rx path offline, Tx path online

- `CANIF_GET_ONLINE`

  Rx and Tx path are switched online

- `CANIF_GET_OFFLINE_ACTIVE`

  Rx and Tx path offline, confirmation is emulated by the CAN Interface

- CANIF_GET_OFFLINE_ACTIVE_RX_ONLINE

  Rx path online, Tx path offline, confirmation is emulated by the CAN Interface

If parameter CanIfPnWakeupTxPduFilterSupport (s. chapter 2.16) is enabled then the following two further modes are available:

- CANIF_GET_TX_ONLINE_WAKF

  Rx path offline, tx path online

- CANIF_GET_ONLINE_WAKF

  Rx and Tx path are switched online

The difference to the modes CANIF_GET_ONLINE and CANIF_GET_TX_ONLINE is that the Tx-PDU filter is activated if the PDU mode is changed to one of these two modes. (s. chapter 2.16).

Caution
If one of the modes CANIF_GET_TX_ONLINE_WAKF or CANIF_GET_ONLINE_WAKF is left by calling of CanIf_SetPduMode() with parameter PduModeRequest which equals CANIF_SET_OFFLINE or CANIF_SET_TX_OFFLINE or CANIF_SET_TX_OFFLINE_ACTIVE or CANIF_SET_ONLINE or CANIF_SET_TX_ONLINE then the Tx-PDU Filter is **deactivated**!

The PDU modes can be set via the function CanIf_SetPduMode() and can be retrieved via the function CanIf_GetPduMode().

2.4 Main Functions

The CAN Interface does not provide any main functions.

2.5 Transmission

The transmission of PDUs is only possible if the CAN controller is in the mode CANIF_CS_STARTED/CANIF_GET_ONLINE or CANIF_CS_STARTED/CANIF_GET_TX_ONLINE.

The Tx request has to be initiated by a call of the service function:

```
CanIf_Transmit(<TxPduId>, <PduInfoPtr>);
```

The PDU ID is used (<TxPduId>) to acquire the PDU properties from the generated data (e.g. responsible CAN Driver, CAN Controller and Can[XL]HardwareObject, CAN ID, CAN XL parameters, …). With this information the CAN Interface prepares the PDU for transmission and then forwards the PDU to the responsible CAN Driver via a call of the service function Can[XL]_<VendorId>_<VendorApiInfix>_Write().

After a successful transmission of the PDU on the CAN bus the confirmation callback function CanIf_TxConfirmation() is called by the CAN Driver either from interrupt context or in case of Tx polling from task context.

This confirmation is dispatched in the CAN Interface to notify the corresponding higher layer about the succesful transmission of the PDU. For this purpose for each PDU a callback function has to be specified at configuration time.

2.5.1 Tx PDU truncation

The feature Tx PDU truncation can be configured for each Tx PDU individually via the parameter `CanIfTxPduTruncation` and can be reconfigured in the Post-build-loadable configuration phase.

If the feature is enabled and the length of a PDU that shall be transmitted exceeds the configured length of the PDU referenced by the PDU ID, the PDU length will be truncated to the configured length.

If the feature is disabled and the length of a PDU that shall be transmitted exceeds the configured length of the PDU referenced by the PDU ID, the transmission request will be rejected. Additionally the runtime error `CANIF_E_TXPDU_LENGTH_EXCEEDED` is reported.

2.5.2 Dynamic transmission

The feature is activated by the parameter “Dynamic Tx Objects”.

The adjustments for the dynamic objects are the same as for the static with the exception that the CAN ID and the attribute whether extended or standard CAN ID can be selected manually.

By default the dynamic object has the CAN ID parameterized during configuration time until it is changed by the call of the API `CanIf_SetDynamicTxId()`. In order to set an extended CAN ID the most significant bit of its value passed to the API shall be set.

The PDU IDs of the dynamic objects are represented as symbolic handles in the file `CanIf_Cfg.h`.

2.5.3 Transmit-buffer

The CAN Interface provides a mechanism to buffer CAN 2.0/FD Tx PDUs (including data) which are mapped to a Tx buffer. This means if the CanHardwareObject of such Tx PDU is occupied, the Tx PDU instance is stored within the CAN Interface until the CanHardwareObject becomes free.

Two handling types of a transmit-buffer are supported:

1. FIFO
2. Prioritization by CAN-identifier

The handling type defines in which manner the Tx-PDUs stored within the Tx-buffer are transmitted in case of the underlying CanHardwareObject becomes free. At all the Tx-PDUs stored within a Tx-buffer are processed either in context of the Tx-confirmation interrupt or in context of CAN Driver’s Tx-main-function in case of polling mode.

The configuration of multiple transmit-buffers is described in chapter 2.5.3.3.

2.5.3.1 FIFO

The stored Tx-PDUs are transmitted in manner First-In-First-Out. Each Tx-PDU-instance is stored. If the FIFO is full then NO Tx-PDUs are stored until the FIFO becomes free.

Caution
If the CAN Driver rejects a transmission request from the CAN Interface for a Tx-PDU out of a FIFO (`Can_Write()` API returns with `CAN_NOT_OK`) or does not confirm a transmitted Tx-PDU from a FIFO to the CAN Interface (`CanIf_TxConfirmation()` API is not called), it is no longer possible to transmit new Tx-PDUs and already buffered Tx-PDUs which belong to the affected FIFO.

The failed transmission out of the Tx-Buffer is signalized by the error code `CANIF_E_TX_BUFFER_TRANSMIT`, if development error reporting is enabled. Additionally the rejection from the CAN Driver can be detected, if the `Can_Write()` API reports a DET.

This situation can be solved by restarting the respective CAN controller (mode transition from STARTED to STOPPED and again to STARTED), which clears all Tx-PDUs from the FIFO.

Caution
In case of transmit-buffer of handling type FIFO only one instance of a Tx-PDU (the last one stored within the FIFO) can be and is cancelled from the FIFO via usage of API `CanIf_CancelTransmit`! (Feature: “Cancellation of Tx-PDUs”, see chapter 4.1.6).

2.5.3.2 Prioritization by CAN-identifier

The stored Tx-PDUs are transmitted in manner: Tx-PDU with high priority is sent before those one with lower priority. The priority is given by the CAN-identifier of the Tx-PDU. A Tx-PDU with a low CAN-identifier has higher priority than a one with greater CAN-identifier. The priority of a Tx-PDU is static and is determined from values of parameters `CanIfTxPduCanId` and `CanIfTxPduCanIdType`. Please consider this aspect in case of configuration of Tx-PDUs with dynamic CAN-identifier. Only one instance of each Tx-PDU is stored within such Tx-buffer: If a Tx-PDU is requested to be transmitted and the Tx-buffer of this Tx-PDU is already in use the already stored data of this Tx-PDU is overwritten in order to ensure the transmission of most newest data.

This handling type can be used to avoid inner priority inversion. This means if the CAN Interface passes a transmit request to the CAN Driver while all CanHardwareObjects are occupied and at least one CanHardwareObject is occupied by a CAN message with lower priority than the message used for the current transmit request the CAN Driver initiates the cancellation of the message with the lowest priority. The cancelled CAN-message is stored in the Tx-buffer of the CAN Interface if the corresponding Tx-buffer is free. Otherwise it is discarded to ensure the transmission of most newest data. By this way a CanHardwareObject becomes free and allows the CAN Interface to pass the CAN-message with the highest priority to the CAN Driver.

Caution
The described: “inner priority inversion” is only supported if at most only one Tx-buffer of handling type: Prioritization by CAN-identifier is configured per CAN controller and hardware transmit cancellation for the CAN Driver is enabled (`CanIfCtrlDrvCfg/CanIfCtrlDrvTxCancellation`).

2.5.3.3 Multiple Transmit-buffers

This feature can only be used if the underlying CAN Driver supports the feature “Multiple Tx-BasicCAN hardware objects”. The Figure 2-1 shows the objects which are needed to be configured within the EcuC-configuration and the relationship among themselves. For each Transmit-buffer a triple of objects: `CanHardwareObject`, `CanIfHthCfg` and `CanIfBufferCfg` must be configured and linked with each other and to corresponding CAN controller (objects: `CanController` and `CanIfCtrlCfg`).

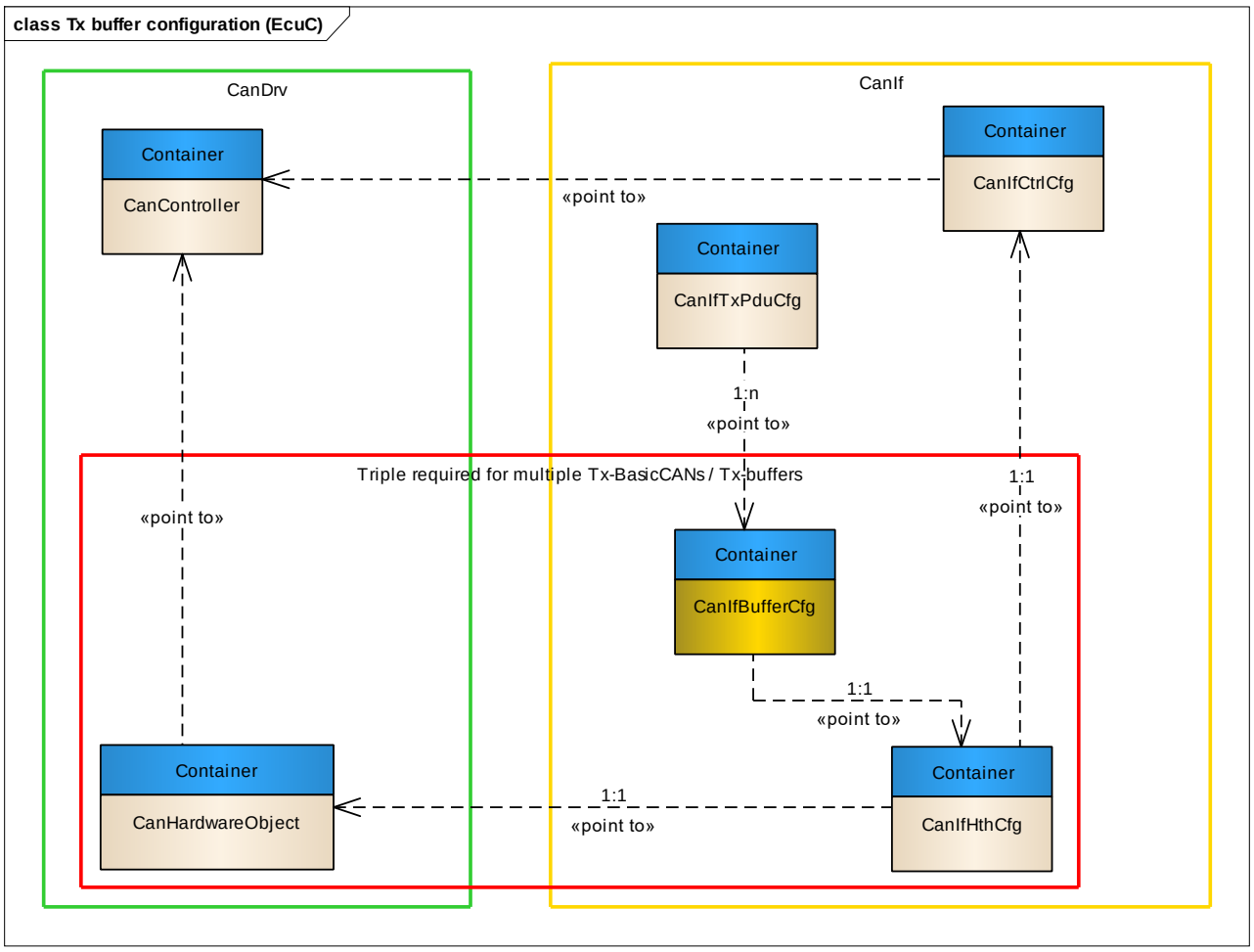

Figure 2-1 Configuration of multiple Transmit-buffers

After this step you can map Tx-PDUs to configured Transmit-buffer (object: CanIfBufferCfg). The described handling type of a transmit-buffer (see chapter 2.5.3) can be configured via the parameter CanIfTxBufferHandlingType. For further information about configuration of a Transmit-buffer please refer to the help which can be found in the GUI of the DaVinci Configurator Classic and to the descriptions of attributes of container CanIfBufferCfg.

2.5.4 Tx confirmation polling support

The CAN Interface supports a service which provides the information on whether any Tx confirmation has occurred for a CAN controller since the last start of that CAN controller at all. This feature can be enabled via the parameter CanIfPublicTxConfirmPollingSupport. If enabled the API CanIf_GetTxConfirmationState() is provided and can be used for this service.

2.5.5 Data checksum Tx

This feature can be used to append a checksum to data of a Tx-PDU. The configuration of such Tx-PDU can be done individually via the parameter CanIfTxPduDataChecksumPdu. The appending of checksum is application specific and must be implemented within the API CanIf_TransmitSubDataChecksumTxAppend(). For further information please see the description of the prototype of this API in chapter 4.4.2.2.

For further information about configuration of this feature at all please refer to the help which can be found in the GUI of the DaVinci Configurator Classic and to the description of mentioned parameters.

2.6 Reception

Reception of PDUs is only possible in the states

- CANIF_CS_STARTED and CANIF_GET_ONLINE

or

- CANIF_CS_STARTED and CANIF_GET_RX_ONLINE.

In all other states the PDUs received by the CAN Driver are discarded by the CAN Interface without notification to the upper layers.

In case of a succesful reception, the upper layer is notified about the PDU ID given by the corresponding upper layer at configuration time, the received data and depending on the used indication function about the length of the received data and the CAN ID.

2.6.1 Reception of CAN 2.0/FD Rx PDUs

The CAN Driver indicates the reception of a CAN 2.0/FD Rx PDU to the CAN Interface by calling the callback function:

```
CanIf_RxIndication(<Mailbox>, <PduInfoPtr>);
```

For CAN 2.0/FD Rx PDUs the CAN Interface supports the reception in Full-CAN and Basic-CAN hardware objects. The upper layers do not notice any differences between these two reception sources as in both cases a callback function is called which was configured for the specified Rx PDU in the generation tool.

In case of a Basic-CAN reception the CAN Interface has to search through a list of all known CAN 2.0/FD Rx PDUs and compare the received CAN ID with the CAN ID in the Rx PDU list.

The CAN Interface offers two different search algorithms for Basic-CAN CAN2.0/FD Rx PDUs with single CAN ID:

- **Linear search**: The list of all CAN 2.0/FD Rx PDUs is searched from high priority (Low CAN ID) to low priority (High CAN ID). This algorithm is efficient for a small amount of CAN 2.0/FD Rx PDUs.
- **Double Hash search**: The CAN 2.0/FD Rx PDU is calculated via two special hash functions. The algorithm is very efficient for a high amount of CAN 2.0/FD Rx PDUs and always takes the same time.

Note
The Double Hash search algorithm uses the mathematical operation modulo.

The search algorithm for Basic-CAN CAN 2.0/FD Rx PDUs with CAN ID range is always linear.

The CAN Interface further applies the following priority order for checking a valid reception in a Basic-CAN at one CAN controller depending on the type of a CAN 2.0/FD Rx PDU.

- CAN 2.0/FD Rx PDU with a single CAN ID
- CAN 2.0/FD Dynamic Rx PDU (see also chapter 2.6.2)
- CAN 2.0/FD Rx PDU with a CAN ID range (Rx Range PDU – see also chapter 2.6.4)

Priority	BasicCAN CAN 2.0/FD Rx PDU
highest	Rx PDU with the lowest single CAN ID[1]
	…
	Rx PDU with the highest single CAN ID[1]
	Dynamic Rx PDU with the smallest CAN ID range
	…
	Dynamic Rx PDU with the largest CAN ID range
	Rx Range PDU with the smallest CAN ID range
	…
lowest	Rx Range PDU with the largest CAN ID range

Table 2-4 Priority order of BasicCAN CAN 2.0/FD Rx PDUs

[1] The CAN IDs are compared left-justified (incl. a possible EXT-ID bit), as is done in the arbitration.

If the CAN ID of an Rx PDU with a single CAN ID is contained in the CAN ID range of a Rx Range PDU, the Rx PDU with the single CAN ID is always checked first for a valid reception.

If the CAN ID of a dynamic Rx PDU is contained in the CAN ID range of a Rx Range PDU, the dynamic Rx PDU is always checked first for a valid reception.

If different Rx Range PDUs or different dynamic Rx PDUs have overlapping CAN ID ranges, the PDU with the smaller CAN ID range is always checked first for valid reception.

If there are two Rx PDUs with the same single CAN ID or two Rx Range PDUs with the same CAN ID range, the Rx PDU with the CAN ID type “CAN 2.0 frame” is always checked first for a valid reception.

If there are multiple Rx Range PDUs with the same CAN ID range size (accept the same number of CAN IDs) or multiple dynamic Rx PDUs with the same CAN ID range / CAN ID range size, the Rx PDUs are checked with the following descending priority:

- Rx PDU with CAN ID type “CAN 2.0 or CAN FD frame”
- Rx PDU with CAN ID type “CAN 2.0 frame”
- Rx PDU with CAN ID type “CAN FD frame”
- Rx PDU name (if CAN ID types are the same)

2.6.2 Reception of CAN XL Rx PDUs

The CAN Driver indicates the reception of a CAN XL Rx PDU to the CAN Interface by calling the callback function:

```
CanIf_XLRxIndication(<Mailbox>, <PduInfoPtr>);
```

The CAN Interface linearly searches through a list of all known CAN XL Rx PDUs and compares the received CAN XL Rx PDU to the configured CAN XL Rx PDUs.

Within this linear search the CAN Interface adheres to the priority order for CAN XL Rx PDUs specified in [4]. For multiple CAN XL Rx PDUs of SDU type 3 which have the same priority according to [4], the same priority order as depicted in Table 2-4 for Basic-CAN CAN 2.0/FD Rx PDUs is applied additionaly.

Caution

The user must ensure that all CAN XL Rx PDUs are distinguishable by the described priority order.

For CAN XL Rx PDUs of SDU type other than 3, this means two PDUs must differ in at least one CAN XL parameter (Acceptance Field, Priority ID, SDU type, VCID).

For CAN XL Rx PDUs of SDU type 3, this means two PDUs must either differ in at least one of the configurable CAN XL parameters (Priority ID, VCID) or in the configured CAN ID, CAN ID range or CAN ID type (“CAN 2.0 frame” or “CAN FD frame”).

2.6.3 Dynamic Rx PDU

For a dynamic Rx PDU the CAN ID of the CAN frame to be received is set during the initialization via a callout function and not at configuration time.

This PDU type is configured by setting the parameter `CanIfRxPduCfg/CanIfRxPduType` to `DYNAMIC` and defining the CAN ID range of the to be set CAN ID by the parameters `CanIfRxPduCfg/CanIfRxPduCanIdRange/CanIfRxPduCanIdRangeLowerCanId` and `CanIfRxPduCfg/CanIfRxPduCanIdRange/CanIfRxPduCanIdRangeUpperCanId`.
It is allowed to configure multiple dynamic Rx PDUs with the same or overlapping CAN ID ranges.

During `CanIf_Init` a configurable callout function (parameter `CanIfDispatchCfg/CanIfDispatchUserSetDynamicRxIdName)` is called for each dynamic Rx PDU. For further information please see the description of the prototype of this callout function in chapter 4.4.2.3

Caution
The user must ensure that a CAN ID for a dynamic Rx PDU is within the configured CAN ID range and is not used for another dynamic Rx PDU or for a Rx PDU with single CAN ID at a CAN controller.

Exception:
A dynamic Rx PDU and a Rx PDU with single CAN ID or another dynamic Rx PDU can have the same CAN ID at one CAN controller, if one Rx PDU has CAN ID type “CAN 2.0 frame” and the other one has CAN ID type “CAN FD frame”.

2.6.4 Rx Range PDU

The Rx Range PDU can be used to receive groups of CAN messages called ranges. A range can be defined either by an upper and a lower CAN ID or by a mask and a code.

The definition of a range by an upper and a lower CAN ID is performed by the following parameters:

- `CanIfRxPduCanIdRangeLowerCanId` and
- `CanIfRxPduCanIdRangeUpperCanId`.

A mask-code-range is defined by the parameters:

- `CanIfRxPduCanId` (code) and
- `CanIfRxPduCanIdMask` (mask).

In case of a mask-code-range each CAN ID which fulfills the following equation pass the range and the reception of the corresponding Rx-PDU is reported to the upper layer.

- `<CAN identifier> & <mask> == <code> & <mask>`

One PDU ID is assigned to all CAN messages which pass the configured range. For each range an indication function can be assigned in the generation tool in order to notify the higher layer about the reception of a CAN message.

A range defined by an upper and a lower CAN ID can be converted into a mask-code-range. Therefore please see the following example.

Example: How to convert a lower CAN ID and an upper CAN ID into mask and code?

Lower CAN ID: 0x400
Upper CAN ID: 0x43F
The code is same as the lower CAN ID:
code = 0x400

You need the count which is upper CAN ID – lower CAN ID → 0x43F – 0x400 = 0x3F
The count 0x3F is 000 0011 1111b in 11-bit binary format. For a range with extended CAN IDs the count needs to be 29-bit wide.

The mask is calculated out of negated count and a 11-bit mask:
mask = ~0x3F & 0x7FF = 0x7C0
For extended IDs you need a 29-bit mask:
mask = ~0x3F & 0x1FFF FFFF = 0x1FFF FFC0

Note:
If for count the first set bit is followed by unset bits on lower significant positions for the calculation of the mask these bits need to be set. For example a count of 0xA3 (1010 0011b) you need to calculate with the count 0xFF (1111 1111b). The consequence is that more CAN IDs are received as intended.

2.6.5 DLC check

The DLC check is executed for all received Rx PDUs after the corresponding configured Rx PDU was identified. The general use of the feature DLC check can only be activated at Pre-compile time. If the feature DLC check is generally activated, it can be configured indidually for each Rx PDU whether the DLC check is carried out. This individual configuration per Rx PDU can be reconfigured in the Post-build-loadable configuration phase.

The DLC check verifies if the received DLC is greater or equal to the DLC specified during configuration time. If the DLC is less than the configured one a DET error is raised and the reception of the PDU is abandoned.

2.6.6 Data checksum Rx

This feature can be used to verify the validity of a Rx-PDU after reception. The Rx-PDU which shall be verified can be configured individually via the parameter `CanIfRxPduDataChecksumPdu`. The verification is application specific and must be implemented within the API `CanIf_RxIndicationSubDataChecksumRxVerify()`. For further information please see the description of the prototype of this API in chapter 4.4.2.1.

In addition an indication function may be configured which signals about invalidity of a Rx-PDU. This indication function can be configured via the parameter `CanIfDispatchDataChecksumRxErrorIndicationName`.

The call of this indication function is application specific too and if required must be invoked within the implementation of `CanIf_RxIndicationSubDataChecksumRxVerify()`.

The prototype of the indication function must match following signature:

- `void My_DataChecksumRxErrFct (PduIdType CanIfRxPduId)`

and can be accessed via the macro: `CanIf_GetDataChecksumRxErrFctPtr()` (see file `CanIf_Cfg.h`)

It is recommended to call this indication function with the identifier of affected Rx-PDU. Therefor the value of parameter `CanIfRxPduId` should be used which is passed by call of `CanIf_RxIndicationSubDataChecksumRxVerify()`. The value of this parameter is a CAN interface internal identifier which corresponds to value of configuration parameter `CanIfRxPduId`. Corresponding macros are generated per Rx-PDU into file `CanIf_Cfg.h`. These ones can be used by application (s. example below).

```
/**********************************************************************************
  \def  AUTOSAR Rx PDU handles
**********************************************************************************/

#define CanIfConf_CanIfRxPduCfg_RxRange2_0                                      0U
#define CanIfConf_CanIfRxPduCfg_RxRange1_0                                      1U
#define CanIfConf_CanIfRxPduCfg_RxMSG00000711_0                                 2U
#define CanIfConf_CanIfRxPduCfg_RxMSG95555311_0                                 3U
#define CanIfConf_CanIfRxPduCfg_RxMSG00000511_0                                 4U
#define CanIfConf_CanIfRxPduCfg_RxMSG91111151_0                                 5U
```

For further information about configuration of this feature at all please refer to the help which can be found in the GUI of the DaVinci Configurator Classic and to the description of mentioned parameters.

2.6.7 Control of reception mode of a Rx-PDU

This feature provides the ability to control the reception mode of a Rx-PDU at runtime. The reception mode can be set per Rx-PDU individually at runtime via the API: `CanIf_SetPduReceptionMode()`. In order to address a Rx-PDU you can use the corresponding symbolic name value which can be found in file `CanIf_Cfg.h` (s. example below).

```
/**********************************************************************************
  \def  AUTOSAR Rx PDU handles
**********************************************************************************/

#define CanIfConf_CanIfRxPduCfg_RxRange2_0                                      0U
#define CanIfConf_CanIfRxPduCfg_RxRange1_0                                      1U
#define CanIfConf_CanIfRxPduCfg_RxMSG00000711_0                                 2U
#define CanIfConf_CanIfRxPduCfg_RxMSG95555311_0                                 3U
#define CanIfConf_CanIfRxPduCfg_RxMSG00000511_0                                 4U
#define CanIfConf_CanIfRxPduCfg_RxMSG91111151_0                                 5U
```

For further information about this API please see chapter 4.1.30. This feature can be used for e.g. either to receive a CAN-message as a Rx-PDU with an explicit CAN-identifier or as a Rx-range-PDU. In case of the configured CAN-identifier of a Rx-PDU fits the range of CAN-identifiers of a Rx-range-PDU on the same CAN controller as well. In case of a FullCAN-Rx-PDU the reception can be controlled at runtime at all.

This feature can be enabled via the parameter `CanIfSetPduReceptionModeSupport`. In addition Rx-PDUs whose reception mode is intended to be controlled at runtime must be configured accordingly via the parameter `CanIfRxPduSetReceptionModePdu`.

For further information about configuration of this feature at all please refer to the help which can be found in the GUI of the DaVinci Configurator Classic and to the description of mentioned parameters.

2.7 Polling

The CAN Interface can process events in polling and interrupt mode. As the polling of events is executed by other layers (e.g. CAN Driver, CAN Transceiver Driver) the CAN Interface is notified by call back functions which are called in the corresponding context.

Note
There is no need for changes in the configuration to run the CAN Interface in polling mode.

2.8 CAN FD

The CAN Interface supports CAN FD. The configuration can be performed both for Rx and Tx PDUs. Therefore please configure the attribute `CanIfRxPduCanIdType` (Rx PDU) and `CanIfTxPduCanIdType` (Tx PDU) accordingly as required by your application.

In case of Rx PDUs the CAN ID type (“CAN 2.0 frame” or “CAN FD frame”) is evaluated during the Rx search algorithm. Hence it is possible to handle two Rx PDUs with the same CAN identifier, at which one is configured as “CAN FD frame” and one as “CAN 2.0 frame”.

Expert Knowledge
If you intend to switch the baudrate of the CAN hardware at runtime it is suggested to use the API `CanIf_SetBaudrate` instead of `CanIf_ChangeBaudrate`.

Rx and Tx FD PDUs with up to 64 bytes payload are supported.

Basic Knowledge
If you intend to configure Basic-CAN CAN FD Tx PDUs and the Tx buffer is enabled in your configuration please ensure that attribute `CanIfStaticFdTxBufferSupport` is enabled.

2.9 CAN XL

The CAN Interface supports the reception and transmission of CAN XL PDUs. A CAN XL PDU is configurable by adding the container `CanIfRxPduXLParams`/`CanIfTxPduXLParams` to a Rx/Tx PDU container.

Caution
The parameter
`CanIf/CanIfInitCfg/CanIfInitHohCfg/CanIfHthCfg/CanIfHthIdSymRef`
or
`CanIf/CanIfInitCfg/CanIfInitHohCfg/CanIfHthCfg/CanIfHrhIdSymRef`
must be manually set per user-define to the corresponding
`AUTOSAR/EcucDefs/Can/CanConfigSet/CanXLHardwareObject`.

The user must manually ensure that a CAN XL PDU and the mapped CanXLHardwareObject belongs to the same CAN controller.

The following features are not supported by a CAN XL Tx PDU and must be disabled:

- Meta data
- Data checksum
- Pn Filter
- PDU type „Dynamic“

The following features are not supported by a CAN XL Rx PDU and must be disabled:

- Meta data
- Data checksum
- Reception mode
- PDU type „Dynamic“

The following features cannot handle CAN XL PDUs:

- J1939 Dynamic Address (do not configure J1939 at a CAN Controller with mapped CAN XL PDUs)
- Bus Mirroring (do not enable Bus Mirroring for a CAN Controller with mapped CAN XL PDUs via CanIf_EnableBusMirroring() during runtime)
- Software transmit cancellation (do not call `CanIf_CancelTransmit()` for a CAN XL Tx PDU)

CAN XL can not be used in combination with:

- Hardware transmit cancellation
- Multi-partition

2.10 Meta data Rx- / Tx-support

If this feature is enabled the CAN Interface supports the handling of dynamic CAN-identifiers by using of SDU meta data. Such dynamic PDU can be configured by parameter `MetaDataLength`. This parameter can be found in the container of corresponding global PDU. In case of configuration variant Post-build loadable please enable this feature by setting of parameter `CanIfMetaDataSupport` to true. In case of configuration variant Pre-compile the activation/deactivation of this feature is determined from the configuration of Rx- and Tx-PDUs. If there is any PDU which has configured the parameter `MetaDataLength` then this feature is enabled else disabled.

2.11 J1939 dynamic address support

If this feature is enabled the CAN Interface translates the addresses (CAN identifiers) of Rx- and Tx-PDUs according to J1939 by using of dynamic address lookup tables. These tables are maintained by J1939Nm by using of following APIs:

- `CanIf_SetAddressTableEntry` and
- `CanIf_ResetAddressTableEntry`.

This feature has to be configured for each CAN controller individually by the parameter `CanIfCtrlJ1939DynAddrSupport`. Please consider that in case of configuration variant Post-build loadable and configuration phase Post-build the value which you can select by `CanIfCtrlJ1939DynAddrSupport` is limited by value of `CanIfJ1939DynAddrSupport` which was set at configuration phase Pre-compile.

Caution

The feature `J1939 dynamic address support` works only if all Rx PDUs of the CAN controller at which this feature is enabled are configured as BasicCANs.

The corresponding hardware filters must be opened for at least:

- the last byte for standard CAN IDs, if `J1939 dynamic address support` is set to `MIXED_CANID`.
- the last two bytes for extended CAN IDs, if `J1939 dynamic address support` is set to `EXTENDED_CANID` or `MIXED_CANID`.

Therefore in case of configuration variant Post-build loadable please first enable this feature as far as you need at all by the parameter `CanIfJ1939DynAddrSupport` and then configure the controller specific parameter of this feature. In case of configuration variant Pre-compile it is only possible to configure the controller specific parameter.

2.12 CAN transceiver handling

The CAN Interface provides APIs and call back functions to control as many CAN transceivers as CAN controllers are available in the system. The CAN transceiver handling has to be activated at Pre-compile time.

The CAN Interface provides the following functions for higher layers to control the behavior of the CAN transceiver.

- `CanIf_SetTrcvMode()`
- `CanIf_TrcvModeIndication()`
- `CanIf_GetTrcvMode()`
- `CanIf_GetTrcvWakeupReason()`
- `CanIf_SetTrcvWakeupMode()`

Additionally the following APIs are provided in order to control a partial networking CAN transceiver.

- `CanIf_CheckTrcvWakeFlag()`
- `CanIf_CheckTrcvWakeFlagIndication()`
- `CanIf_ClearTrcvWufFlag()`
- `CanIf_ClearTrcvWufFlagIndication()`
- `CanIf_ConfirmPnAvailability()`

The initialization of the CAN Transceiver Driver itself is not executed by the CAN Interface. This means the calling layer has to make sure the CAN Transceiver Driver is initialized before using the listed API functions.

If more than one different CAN Transceiver Driver is used in the system the CAN Interface provides a mapping to address the correct CAN Transceiver Driver with the correct parameters. The parameter `CanIfTransceiverMapping` has to be activated to control more than one CAN Transceiver Driver.

It is also allowed to activate the parameter `CanIfTransceiverMapping` if only one CAN Transceiver Driver is used in the system. Because of additional runtime it is suggested to deactivate this feature in this use case.

The CAN Interface supports the detection of wakeup events raised by a CAN transceiver. The feature “Wakeup Support” has to be activated and a wakeup source has to be configured for the corresponding CAN transceiver.

Within the API `CanIf_CheckWakeup()` the CAN Interface analyses the passed wakeup source parameter and decides whether a CAN controller or a CAN transceiver has to be requested for a pending wakeup event.

For more details refer to the chapter 2.13 Sleep / Wakeup.

2.13 Sleep / Wakeup

The CAN Interface controls the modes of the underlying CAN Driver and CAN Transceiver Driver.

The API `CanIf_SetControllerMode()` has to be used to change the mode of the CAN controller while the CAN transceiver can be controlled with the API `CanIf_SetTrcvMode()`.

Caution
The CAN Interface itself does not perform any checks whether the CAN controller and the CAN transceiver are set to sleep consistently and in the correct sequence. It is up to the higher layer to call `CanIf_SetControllerMode()` and `CanIf_SetTrcvMode()` in the correct sequence.

Wakeup events can be raised either by the CAN controller or by the CAN transceiver. In both cases the CAN Interface is not directly informed about state changes. This means the higher layers (normally the EcuM) has to call the API `CanIf_CheckWakeup()` with the wakeup sources configured for CAN transceiver or CAN controller (1).

The CAN Interface decides by analyzing the passed wakeup source whether the CAN controller or the CAN Transceiver Driver has to be checked for a pending wakeup (2 or 2').

The following figure illustrates the described wakeup sequence:

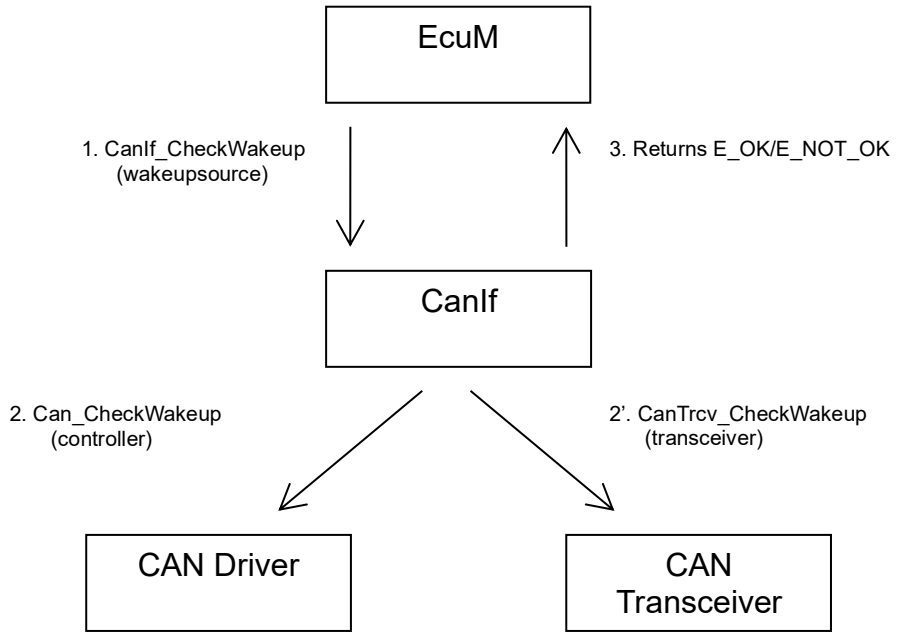

Figure 2-2 Wakeup sequence (No validation)

If the parameter "CanIfPublicWakeupCheckValidSupport" is enabled the following figure shows the sequence which has to be executed for a valid wakeup. Steps 1 to 3 take place as described above.

After the call of EcuM_SetWakeupEvent() the CAN Interface has to be set to the state CANIF_CS_STARTED to be able to receive messages. These messages won't be passed to upper layers by the CAN Interface because the PDU-mode is still set to OFFLINE. The state change which sets the CAN Interface to the mode STARTED has to be realized by the call of the API CanIf_SetControllerMode() with mode CANIF_CS_STARTED (5) from the function EcuM_StartWakeupSources() (4). If the wakeup was detected by the CAN transceiver the CAN controller has to be woken up internally. This means the call CanIf_SetControllerMode() with mode CANIF_CS_STOPPED is necessary in (5) before the transition to mode STARTED is executed.

If the wakeup is initiated by the CAN controller the corresponding CAN transceiver has to be set to mode NORMAL and the CAN controller has to be set to mode STARTED.

If the wakeup is initiated by a CAN transceiver the CAN controller has to be woken up internally. This means an additional call of CanIf_SetControllerMode() with mode CANIF_CS_STOPPED has to be executed to wakeup the CAN controller before the transition to mode STARTED is initiated. (Depending on the behavior of the CAN transceiver the CAN controller and the configuration itself it is possible to wakeup both the CAN controller and the CAN transceiver externally.)

Next the EcuM starts a time out for the wakeup validation. This means if a message is received within this timeout (6) the call of CanIf_CheckValidation() executed by the EcuM (7) will result in a successful validation. The CAN Interface checks for a recent Rx event (6) which occurred after the wakeup and notifies the EcuM by calling of EcuM_ValidationWakeupEvent().

If there is no message reception after (5) the function CanIf_CheckValidation() has been called no successful wakeup validation won't be notified and the EcuM will run into a timeout. In this case the EcuM calls EcuM_StopWakeupSources() (8') and the CAN Driver and CAN transceiver have to be set to mode SLEEP again.

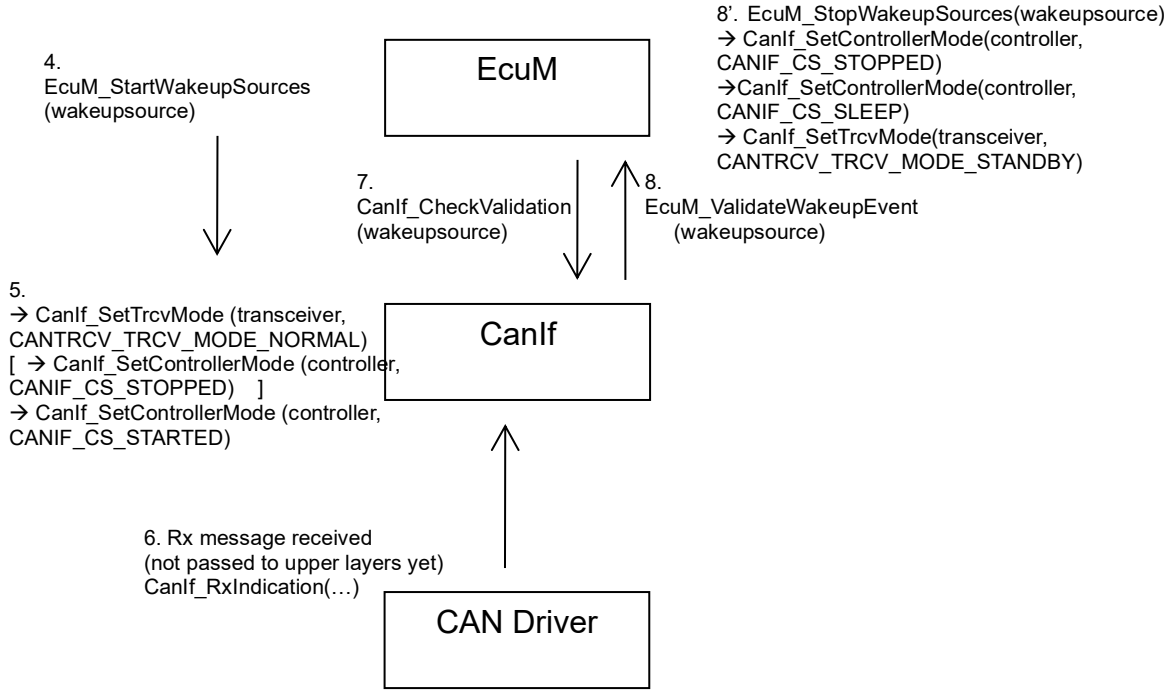

Figure 2-3 Wakeup sequence (Wakeup validation)

During the wakeup sequence as well as during the transition to mode `SLEEP`, the higher layers have to take care about the sequence of the state transitions affecting the CAN controller (CAN Driver) and the CAN Transceiver Driver.

Since ASR4.0R3 it is configurable on whether only a received CanNm-message is able to do the validation.

2.14 Bus Off

The CAN Interface handles bus off events notified by the CAN Driver in interrupt driven or polling systems. If a bus off event is raised the CAN Driver forwards it to the CAN Interface by calling the function `CanIf_ControllerBusOff()`.

The CAN Interface switches its internal CAN controller state from `STARTED` to `STOPPED` and the PDU mode is set to `OFFLINE`. In this state no reception and no transmission is possible until the CAN Interface's controller state and as a result the CAN controller's bus off state is recovered by the call of the function `CanIf_SetControllerMode()` for the affected CAN controller by the higher layer.

After the CAN controller state is switched the bus off state is recovered. For successful reception and transmission the PDU mode has to be switched to `RX_ONLINE`, `TX_ONLINE` or `ONLINE` by the higher layer.

2.15 Version Info

The version of the CAN Interface module can be acquired in three different ways. The first possibility is by calling of the function `CanIf_GetVersionInfo()`. This function returns the module's version in the structure `Std_VersionInfoType` which includes the VendorID and the ModuleID additionally.

The second possibility is the access of version defines which are specified in the header file `CanIf.h`.

The following defines can be evaluated to access different versions:

AUTOSAR version:

- `CANIF_AR_RELEASE_MAJOR_VERSION`
- `CANIF_AR_RELEASE_MINOR_VERSION`
- `CANIF_AR_RELEASE_PATCH_VERSION`

Module version:

- `CANIF_SW_MAJOR_VERSION`
- `CANIF_SW_MINOR_VERSION`
- `CANIF_SW_PATCH_VERSION`

Module ID:

- `CANIF_MODULE_ID`

Vendor ID:

- `CANIF_VENDOR_ID`

There is a third possibility to at least acquire the SW version by accessing globally visible constants:

- `CanIf_MainVersion`
- `CanIf_SubVersion`
- `CanIf_ReleaseVersion`

Note
The API `CanIf_GetVersionInfo()` is only available if enabled at Pre-compile time. The definitions can be accessed independent of the configuration.

2.16 Partial Networking

This feature consists of two sub-features:

- Wakeup Tx-PDU filter (parameter: `CanIfPnWakeupTxPduFilterSupport`)
- Handling of a partial networking CAN transceiver (parameter: `CanIfPnTrcvHandlingSupport`)

The mentioned sub-features can be used only if the attribute `CanIfPublicPnSupport` is enabled. See the following table for more information about mentioned sub-features.

Feature	Description
`CanIfPnWakeupTxPduFilterSupport`	Tx-PDU filter which is activated if the PDU mode is changed either to `CANIF_SET_ONLINE_WU_FILTER` or to `CANIF_SET_TX_ONLINE_WU_FILTER`. This filter is active until the first Tx-confirmation / Rx-indication of the corresponding CAN controller arrives. Only certain Tx-PDUs which are labeled as Tx wakeup filter PDUs (s. parameter `CanIfTxPduPnFilterPdu`) can pass the filter. All Tx-requests of other Tx-PDUs are refused by CAN Interface until the filter is disabled.
`CanIfPnTrcvHandlingSupport`	Handling of a partial networking CAN transceiver

Table 2-5 Sub-features of feature Partial Networking

The parameter `CanIfPnTrcvHandlingSupport` is enabled automatically if at least one underlying CAN Transceiver Driver supports partial networking. In case of using the feature `CanIfPnWakeupTxPduFilterSupport` the Tx-PDUs which are allowed to pass the filter have to be configured accordingly. This kind of configuration can be performed individually for every Tx-PDU via the parameter `CanIfTxPduPnFilterPdu`.

Note
Please consider that the filter of a certain CAN controller is only active if at least one Tx-PDU of this CAN controller has the parameter `CanIfTxPduPnFilterPdu` enabled.

The feature `CanIfPnWakeupTxPduFilterSupport` is configurable in both configuration variants:

- Pre-compile
- Post-build-loadable

Except the restriction that this feature has to be enabled at Pre-compile time at all there are no any further restrictions concerning the reconfiguration of this feature in accordance with the Tx-PDUs which may pass the filter in case of a Post-build-loadable configuration variant.

2.17 Bus Mirroring

The Bus Mirroring can be globally enabled at Pre-compile time. If it is globally enabled, the mirroring can be enabled/disabled per CAN controller with the API `CanIf_EnableBusMirroring()` during runtime. After initialization mirroring is disabled on all CAN controllers.

When the mirroring is enabled on a CAN controller for each successfully transmitted or received CAN frame the CAN-ID, length and content is reported to the Bus Mirroring module. All received CAN frames that pass the hardware acceptance filter are reported, regardless of whether the CAN frame (L-PDU) exists in the ECU configuration or not. Transmitted CAN frames are reported when the transmission is confirmed by the MICROSAR Classic CAN Driver callout function `Appl_GenericConfirmation()`.

The service functions to get the current error state, the current Rx error counter and the current Tx error counter from a CAN controller

- `CanIf_CanIf_GetControllerErrorState()`
- `CanIf_GetControllerRxErrorCounter()`
- `CanIf_GetControllerTxErrorCounter()`

are used by the Bus Mirroring module and are only available, if Bus Mirroring is globally enabled.

Caution

A successful transmitted CAN frame is not reported from the callback function `CanIf_TxConfirmation()` to the Bus Mirroring module as specified by AUTOSAR release version 4.4.0 but instead from the MICROSAR Classic CAN Driver callout function `Appl_GenericConfirmation()`. Therefore, it is not necessary to store the content of each CAN frame before it is transmitted in the CAN Interface as specified by AUTOSAR release version 4.4.0. The MICROSAR Classic CAN Driver stores the content of each CAN frame before it is transmitted and provides the stored content after a successful transmission to the CAN Interface by the callout function `Appl_GenericConfirmation()`.

The feature can only be enabled, if only MICROSAR Classic CAN Driver with the BSWMD parameter `CanGeneral/CanGenericConfirmationAPI2` are used.

2.18 Security event reporting

The security event reporting can be enabled at Pre-compile time. If it is enabled, every detected security event at a CAN controller is reported with the associated context data to the Intrusion Detection System Manager module (IdsM).

The following security events with their context data can be reported:

Security event	Context data
CANIF_SEV_ERRORSTATE_BUSOFF	CanIf ControllerId
CANIF_SEV_ERRORSTATE_PASSIVE	CanIf ControllerId and result, which error counter(s) exceeded the threshold
CANIF_SEV_RX_ERROR_DETECTED	CanIf ControllerId and reception CAN error
CANIF_SEV_TX_ERROR_DETECTED	CanIf ControllerId and transmission CAN error

Table 2-6 Security events with their context data

Note
Between the occurrence of an error state passive on a CAN controller and its notification to the CAN interface, the error counters can meanwhile have fallen below the threshold value again. In this case the CAN Interface is notified about the error state passive at a CAN controller with error counter values below the threshold. If both error counter values are below the threshold the *CANIF_SEV_ERRORSTATE_PASSIVE* is reported with the worst-case result in the context data, that both counters exceeded the threshold (`CANIF_RX_TX_THRESHOLD_EXCEEDED`), to the IdsM.

2.19 CAN Drivers with different AR versions

The CAN Interface supports CAN Drivers from third party vendors based on different AUTOSAR release versions which are abstracted by the following equivalence classes (EQCs):

- EQC AR 4.0.3: Third party CAN Driver based on AUTOSAR release version 4.0.3.
- EQC AR 4.2.1: Third party CAN Driver based on AUTOSAR release version 4.2.1, 4.2.2 or 4.3.0.
- EQC AR 4.3.1: Third party CAN Driver based on AUTOSAR release version 4.3.1, 4.4.0, R19-11, R20-11, R21-11 or R22-11.

The equivalence class of a used third party CAN Driver must be configured at Pre-compile time via the CAN Driver specific parameter `CanIf/CanIfCtrlDrvCfg/CanIfCtrlDrvArVersion`.

Note
All third party CAN Drivers in a configuration must have the same equivalence class.

The configured equivalence class affects only the interface between the CAN Interface and the underlying CAN Driver. The interfaces to the upper layer are the same for all equivalence classes (see chapter 4).

The CAN Interface expects the following naming convention for the service functions of a CAN Driver

`Can_<VendorId>_<VendorApiInfix>_<ApiName>`

`CanXL_<VendorId>_<VendorApiInfix>_<ApiName>`

based on the values of the attributes "VendorId" and "VendorApiInfix" from the BSWMD file of the CAN Driver.

The following CAN Driver configurations are supported:

- One MICROSAR Classic CAN Driver
- One third party CAN Driver of EQC AR 4.0.3, AR 4.2.1 or AR 4.3.1
- Multiple MICROSAR Classic CAN Drivers
- Multiple third party CAN Drivers of EQC AR 4.2.1
- Multiple third party CAN Drivers of EQC AR 4.3.1
- One third party CAN Driver of EQC AR 4.0.3 with one or multiple infixed MICROSAR Classic CAN Drivers
- One or multiple third party CAN Driver of EQC AR 4.2.1 with one or multiple infixed MICROSAR Classic CAN Drivers
- One or multiple third party CAN Driver of EQC AR 4.3.1 with one or multiple infixed MICROSAR Classic CAN Drivers

Based on the configured CAN Drivers, the use of the following AUTOSAR release version specific features and MICROSAR Classic features are not supported:

- Hardware transmit cancellation for third party CAN Drivers of EQC AR 4.2.1 or AR 4.3.1.
- Change/check baudrate if at least one third party CAN Driver of EQC AR 4.2.1 or AR 4.3.1 is configured.
- Set baudrate if at least one third party CAN Driver of EQC AR 4.0.3 is configured.
- Extended RAM-check, Software transmit cancellation, Bus Mirroring and Multi-Partition if at least one third party CAN Driver is configured.

2.20 Extended RAM-check

This feature is configured via the parameter `CanIfExtendedRamCheckSupport`. For further information about configuration of this feature please refer to the description of mentioned parameter.

The feature can not be used with third party CAN Drivers.

2.21 Multi-partition

The CAN controllers (`CanIf/CanIfCtrlDrvCfg/CanIfCtrlCfg`) can be mapped to different partitions via the parameter `CanIfCtrlCfg/CanIfEcucPartitionRef`. The underlying MICROSAR Classic CAN Drivers provide the information, which CAN controllers can be mapped individually or together to a partition.

Each partition can be further mapped to a (physical) core. This offers the possibility of load balancing because the processing for the CAN controllers (e.g. rx and tx processing) can then be executed in parallel.

The initialization of the CAN Interface have to be always executed on the main core.

The following configuration constraints have to be considered:

- Each different partition referenced via the parameter `CanIfCtrlCfg/CanIfEcucPartitionRef` has to be mapped to a separate core (at the corresponding `OsApplication` over `OsApplication/OsApplicationCoreRef`).
- All partitions referenced via the parameter `CanIfCtrlCfg/CanIfEcucPartitionRef` have to have the same ASIL level (parameter `EcucPartition/ASIL`).
- CanIf-Rx-PDUs and CanIf-Tx-PDUs have to be mapped to the same partition (over the referenced global PDU) as the corresponding CAN controller.
- The user of a CanIf-PDU has to be mapped to the same partition (except for the PduR as user).
- A wakeup source (referenced by `CanController/CanWakeupSourceRef` or `CanController/CanWakeupSourceId`) has to be partition unique.

Multi-partion can not be used in combination with:

- J1939 dynamic address support
- Security event reporting
- Bus Mirroring
- Third party CAN Drivers

2.22 Error Handling

2.22.1 Development Error Reporting

By default, development errors are reported to the DET using the service `Det_ReportError()` as specified in [2], if development error reporting is enabled (i.e. Pre-compile parameter `CANIF_DEV_ERROR_REPORT==STD_ON`).

If another module is used for development error reporting, the function prototype for reporting the error can be configured by the integrator, but must have the same signature as the service `Det_ReportError()`.

The reported CAN Interface ID is 60.

The reported service IDs identify the services which are described in chapter 4. The following table presents the service IDs and the related services:

Service ID		Service
1	CANIF_INIT_API	CanIf_Init
3	CANIF_SETCONTROLLERMODE_API	CanIf_SetControllerMode
4	CANIF_GETCONTROLLERMODE_API	CanIf_GetControllerMode
5	CANIF_TRANSMIT_API	CanIf_Transmit
9	CANIF_SETPDUMODE_API	CanIf_SetPduMode
10	CANIF_GETPDUMODE_API	CanIf_GetPduMode
11	CANIF_GETVERSIONINFO_API	CanIf_GetVersionInfo
12	CANIF_SETDYNAMICTXID_API_ID	CanIf_SetDynamicTxId
13	CANIF_SETTRCVMODE_API	CanIf_SetTrcvMode
14	CANIF_GETTRCVMODE_API	CanIf_GetTrcvMode
15	CANIF_GETTRCVWAKEUPREASON_API	CanIf_GetTrcvWakeupReason
16	CANIF_SETTRCVWAKEUPMODE_API	CanIf_SetTrcvWakeupMode
17	CANIF_CHECKWAKEUP_API	CanIf_CheckWakeup
18	CANIF_CHECKVALIDATIONUP_API	CanIf_CheckValidation
19	CANIF_TXCONFIRMATION_API	CanIf_TxConfirmation
20	CANIF_RXINDICATION_API	CanIf_RxIndication
21	CANIF_CANCELTXCONFIRMATION_API	CanIf_CancelTxConfirmation
22	CANIF_CONTROLLERBUSOFF_API	CanIf_ControllerBusoff
23	CANIF_CONTROLLERMODEINDICATION_API	CanIf_ControllerModeIndication
24	CANIF_TRCVMODEINDICATION_API	CanIf_TrcvModeIndication
25	CANIF_GETTXCONFIRMATIONSTATE_API	CanIf_GetTxConfirmationState
26	CANIF_CONFIRMPNAVAILABILITY_API	CanIf_ConfirmPnAvailability
27	CANIF_BAUDRATECHANGE_API	CanIf_ChangeBaudrate
28	CANIF_BAUDRATECHECK_API	CanIf_CheckBaudrate
30	CANIF_CLEARTRCVWUFFLAG_API	CanIf_ClearTrcvWufFlag
31	CANIF_CHECKTRCVWAKEFLAG_API	CanIf_CheckTrcvWakeFlag
32	CANIF_CLEARTRCVWUFFLAGINDICATION_API	CanIf_ClearTrcvWufFlagIndication
33	CANIF_CHECKTRCVWAKEFLAGINDICATION_API	CanIf_CheckTrcvWakeFlagIndication
39	CANIF_BAUDRATESET_API	CanIf_SetBaudrate

Service ID		Service
75	CANIF_GETCONTROLLERERRORSTATE_API	CanIf_GetControllerErrorState
76	CANIF_ENABLEBUSMIRRORING_API	CanIf_EnableBusMirroring
77	CANIF_GETCONTROLLERRXERRORCOUNTER_API	CanIf_GetControllerRxErrorCounter
78	CANIF_GETCONTROLLERTXERRORCOUNTER_API	CanIf_GetControllerTxErrorCounter
79	CANIF_CONTROLLERERRORSTATEPASSIVE_API	CanIf_ControllerErrorStatePassive
80	CANIF_ERRORNOTIFICATION_API	CanIf_ErrorNotification
85	CANIF_XLRXINDICATION_API	CanIf_XLRxIndication
243	CANIF_APPL_GENERICCONFIRMATION_API	Appl_GenericConfirmation
244	CAN_IF_RAMCHECKCORRUPTCONTROLLER_API	CanIf_RamCheckCorruptController
245	CAN_IF_RAMCHECKCORRUPTMAILBOX_API	CanIf_RamCheckCorruptMailbox
246	CANIF_SETPDURECEPTIONMODE_API	CanIf_SetPduReceptionMode
247	CANIF_RAMCHECKENABLECONTROLLER_API	CanIf_RamCheckEnableController
248	CANIF_RAMCHECKENABLEMAILBOX_API	CanIf_RamCheckEnableMailbox
249	CANIF_RAMCHECKEXECUTE_API	CanIf_RamCheckExecute
250	CANIF_CANCELTRANSMIT_API	CanIf_CancelTransmit
251	CANIF_TXNOTIFICATION_API	CanIf_CancelTxNotification
252	CANIF_SETADDRESSTABLEENTRY_API	CanIf_SetAddressTableEntry
253	CANIF_RESETADDRESSTABLEENTRY_API	CanIf_ResetAddressTableEntry
254	CANIF_GETTXADDRESSTABLEENTRY_API	CanIf_GetTxAddressTableEntry
255	CANIF_GETRXADDRESSTABLEENTRY_API	CanIf_GetRxAddressTableEntry

Table 2-7 Service IDs

The errors reported to DET are described in the following table:

Error Code		Description
10	CANIF_E_PARAM_CANID	Used in context of following functions if an invalid CAN identifier is passed: - CanIf_RxIndication - CanIf_SetDynamicTxId - CanIf_Init
11	CANIF_E_PARAM_DLC	Used in context of following functions if a PDU with invalid data length is passed: - CanIf_RxIndication - CanIf_XLRxIndication - CanIf_Transmit - CanIf_CancelTxConfirmation
12	CANIF_E_PARAM_HRH	Used in context of following function if an invalid hardware receive handle is passed: - CanIf_RxIndication

Error Code		Description
13	CANIF_E_PARAM_LPDU	Used in context of following functions if an invalid Tx-PDU is passed: - CanIf_TxConfirmation - CanIf_CancelTxConfirmation - CanIf_CancelTxNotification - Appl_GenericConfirmation
14	CANIF_E_PARAM_CONTROLLER	Used in context of following functions if an invalid CAN controller is passed: - CanIf_ControllerBusOff - CanIf_ControllerModeIndication - CanIf_GetTxConfirmationState - CanIf_SetTrcvMode - CanIf_GetTrcvMode - CanIf_GetTrcvWakeupReason - CanIf_SetTrcvWakeupMode - CanIf_TrcvModeIndication - CanIf_ConfirmPnAvailability - CanIf_ClearTrcvWufFlag - CanIf_ClearTrcvWufFlagIndication - CanIf_CheckTrcvWakeFlag - CanIf_CheckTrcvWakeFlagIndication
15	CANIF_E_PARAM_CONTROLLERID	Used in context of following functions if an invalid CAN controller is passed: - CanIf_SetControllerMode - CanIf_GetControllerMode - CanIf_SetPduMode - CanIf_GetPduMode - CanIf_CheckBaudrate - CanIf_ChangeBaudrate - CanIf_SetBaudrate - CanIf_SetAddressTableEntry - CanIf_ResetAddressTableEntry - CanIf_Transmit - CanIf_TxConfirmation - CanIf_CancelTxConfirmation - CanIf_CancelTransmit - CanIf_CheckWakeup - CanIf_RamCheckExecute - CanIf_RamCheckEnableMailbox - CanIf_RamCheckEnableController - CanIf_GetControllerErrorState - CanIf_GetControllerRxErrorCounter

Error Code		Description
15	CANIF_E_PARAM_CONTROLLERID	- CanIf_GetControllerTxErrorCounter - CanIf_EnableBusMirroring - Appl_GenericConfirmation - CanIf_ControllerErrorStatePassive - CanIf_ErrorNotification - CanIf_GetTxAddressTableEntry - CanIf_GetRxAddressTableEntry - CanIf_XLRxIndication
16	CANIF_E_PARAM_WAKEUPSOURCE	Used in context of following functions if an invalid wakeup source is passed: - CanIf_CheckValidation - CanIf_CheckWakeup
17	CANIF_E_PARAM_TRCV	Used in context of following functions if an invalid CAN transceiver is passed: - CanIf_TrcvModeIndication - CanIf_GetTrcvWakeupReason - CanIf_GetTrcvMode - CanIf_SetTrcvMode - CanIf_SetTrcvWakeupMode - CanIf_ConfirmPnAvailability - CanIf_ClearTrcvWufFlagIndication - CanIf_CheckTrcvWakeFlagIndication - CanIf_ClearTrcvWufFlag - CanIf_CheckTrcvWakeFlag - CanIf_CheckWakeup
18	CANIF_E_PARAM_TRCVMODE	Used in context of following function if an invalid CAN transceiver mode is passed: - CanIf_SetTrcvMode
19	CANIF_E_PARAM _TRCVWAKEUPMODE	Used in context of following function if an invalid CAN transceiver wakeup mode is passed: - CanIf_SetTrcvWakeupMode
20	CANIF_E_PARAM_POINTER	Used in context of following functions if an invalid pointer is passed: - CanIf_Init - CanIf_GetControllerMode - CanIf_Transmit - CanIf_RxIndication - CanIf_XLRxIndication - CanIf_GetPduMode - CanIf_GetVersionInfo - CanIf_GetTrcvWakeupReason - CanIf_GetTrcvMode

Error Code		Description
20	CANIF_E_PARAM_POINTER	- CanIf_CancelTxConfirmation - CanIf_GetControllerErrorState - CanIf_GetControllerRxErrorCounter - CanIf_GetControllerTxErrorCounter - Appl_GenericConfirmation
21	CANIF_E_PARAM_CTRLMODE	Used in context of following function if an invalid CAN controller mode is passed: - CanIf_SetControllerMode
22	CANIF_E_PARAM_PDU_MODE	Used in context of following function if an invalid PDU mode is passed: - CanIf_SetPduMode
23	CANIF_E_PARAM_CAN_ERROR	Used in context of following function if an invalid CAN error is passed: - CanIf_ErrorNotification
30	CANIF_E_UNINIT	Used in context of following functions if called before the CAN Interface is initialized: - CanIf_Transmit - CanIf_TxConfirmation - CanIf_RxIndication - CanIf_XLRxIndication - CanIf_ControllerBusOff - CanIf_SetPduMode - CanIf_GetPduMode - CanIf_CancelTxConfirmation - CanIf_CheckWakeup - CanIf_CheckValidation - CanIf_GetTrcvWakeupReason - CanIf_SetTrcvWakeupMode - CanIf_ControllerModeIndication - CanIf_SetDynamicTxId - CanIf_TrcvModeIndication - CanIf_SetControllerMode - CanIf_GetControllerMode - CanIf_CancelTxNotification - CanIf_SetTrcvMode - CanIf_GetTrcvMode - CanIf_CancelTransmit - CanIf_ConfirmPnAvailability - CanIf_ClearTrcvWufFlagIndication - CanIf_CheckTrcvWakeFlagIndication - CanIf_ClearTrcvWufFlag - CanIf_CheckTrcvWakeFlag

Error Code		Description
30	CANIF_E_UNINIT	- CanIf_GetTxConfirmationState - CanIf_CheckBaudrate - CanIf_ChangeBaudrate - CanIf_SetBaudrate - CanIf_SetPduReceptionMode - CanIf_SetAddressTableEntry - CanIf_ResetAddressTableEntry - CanIf_RamCheckExecute - CanIf_RamCheckEnableMailbox - CanIf_RamCheckEnableController - CanIf_SetPduReceptionMode - CanIf_GetControllerErrorState - CanIf_GetControllerRxErrorCounter - CanIf_GetControllerTxErrorCounter - CanIf_EnableBusMirroring - Appl_GenericConfirmation - CanIf_ControllerErrorStatePassive - CanIf_ErrorNotification - CanIf_GetTxAddressTableEntry - CanIf_GetRxAddressTableEntry
44	CANIF_E_INVALID_PDURECEPTIONMODE	Used in context of following function if an invalid reception mode is passed: - CanIf_SetPduReceptionMode
50	CANIF_E_INVALID_TXPDUID	Used in context of following functions if an invalid Tx-PDU is passed: - CanIf_CancelTransmit - CanIf_SetDynamicTxId - CanIf_Transmit - CanIf_TxConfirmation
60	CANIF_E_INVALID_RXPDUID	Used in context of following function if an invalid Rx-PDU is passed: - CanIf_SetPduReceptionMode
61	CANIF_E_INVALID_DLC	Used in context of following functions if the length of received PDU is invalid (smaller than the configured one): - CanIf_RxIndication - CanIf_XLRxIndication
62	CANIF_E_DATA_LENGTH_MISMATCH	Used in context of following function if a CAN XL Tx PDU which shall be transmitted has an invalid data length: - CanIf_Transmit

Error Code		Description
70	CANIF_E_STOPPED	Used in context of following function if it is called while either the controller mode is STOPPED or the PDU mode is OFFLINE: - CanIf_Transmit
71	CANIF_E_NOT_SLEEP	Used in context of following function if it is called while the CAN controller mode is neither in SLEEP nor in STOPPED. - CanIf_CheckWakeup
90	CANIF_E_TXPDU_LENGTH_EXCEEDED	Used to inform that a PDU which shall be transmitted exceeds the configured length. Used in context of following function: - CanIf_Transmit
Additionally defined error codes (not AUTOSAR compliant)		
45	CANIF_E_CONFIG	Used to detect inconsistent data in the generated files due to misconfiguration. Used in context of following functions: - CanIf_RxIndication - CanIf_XLRxIndication - CanIf_Transmit - CanIf_RamCheckCorruptMailbox - CanIf_RamCheckCorruptController - CanIf_RamCheckExecute - CanIf_RamCheckEnableMailbox - CanIf_GetControllerErrorState - CanIf_GetControllerRxErrorCounter - CanIf_GetControllerTxErrorCounter
46	CANIF_E_FATAL	Used to detect either an invalid (out of bounce) write access to a variable or an invalid read access to function pointer tables in order to prevent undefined behaviour at runtime. Used in context of following functions: - CanIf_Init - CanIf_Transmit - CanIf_CancelTxConfirmation - CanIf_CancelTransmit - CanIf_CancelTxNotification - CanIf_TxConfirmation

Error Code		Description
47	CANIF_E_INVALID_SA	Used in context of following functions if an invalid J1939 source address (SA) is determined: - CanIf_Transmit - CanIf_RxIndication
48	CANIF_E_INVALID_DA	Used in context of following functions if an invalid J1939 destination address (DA) is determined: - CanIf_Transmit - CanIf_RxIndication
49	CANIF_E_INVALID_CANIDTYPESIZE	Used in context of following function if the size [bytes] of type Can_IdType is inconsistent between static and generated code: - CanIf_Init
50	CANIF_E_INVALID_DLC_METADATA	Used in context of following function if a Rx-PDU of type: meta data is received with invalid length - CanIf_RxIndication
51	CANIF_E_FULL_TX_BUFFER_FIFO	Used to inform that the transmit-buffer of handling type FIFO is full and that no further Tx-PDUs can be buffered. Used in context of following function: - CanIf_Transmit
52	CANIF_E_INVALID_DOUBLEHASH_CALC	Used in context of following function if the calculated match via the double hash algorithm for a received CAN message is not in valid range: - CanIf_RxIndication
53	CANIF_E_TX_BUFFER_TRANSMIT	Used in context of the following function as information that the transmission out of a Tx-Buffer has failed: - CanIf_Transmit
54	CANIF_E_INVALID_XLPARAM	Used in context of the following function if at least one CAN XL parameter has an invalid value: - CanIf_XLRxIndication

Table 2-8 Errors reported to DET

Caution
If the development error detection is disabled not only the reporting of the errors is suppressed but also the detection i.e. the verification of valid function parameters.

2.22.2 Production Code Error Reporting

The CAN Interface has no specified Production Errors [1]. Therefore, the CAN Interface does not support a Production Code Error Reporting.

3 Integration

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

3.1 Embedded Implementation

The delivery of the CAN Interface contains these source code files:

File Name	Description	Integration Tasks
CanIf.c	This is the source file of the CAN Interface. It contains the implementation of the CAN Interface.	-
CanIf.h	This is the header file of the CAN Interface. It contains the declaration of the public APIs and global data of the CAN Interface. The file has to be included by upper layer modules to use the service functions provided by the CAN Interface.	-
CanIf_Cbk.h	This is the callback function header file of the CAN Interface. It contains the declaration of the callback functions of the CAN Interface. The file has to be included by lower layer modules to access callback functions provided by the CAN Interface.	-
CanIf_Types.h	This is the type defintion header file of the CAN Interface. It contains the definition of types provided by the CAN Interface which have to be used by other modules. This file is included automatically if either CanIf.h or CanIf_Cbk.h is included.	-
CanIf_General Types.h	This is the general type definition header file of the CAN Interface. It contains the public types of the CAN Interface. This header file is included by Can_GeneralTypes.h.	-
CanIf_MemMap.h	Generated header file that contains the memory mapping sections of the CAN Interface. This file is included automatically by CanIf.h, CanIf.c, CanIf_Cbk.h, CanIf_Cfg.h, CanIf_Lcfg.c and CanIf_PBcfg.c.	-
CanIf_Cfg.h	Generated header file that contains public constant data, external data declarations, type definitions and abstracted access to the generated data and variables. The file is included automatically by CanIf.h and CanIf_Cbk.h.	-
CanIf_Lcfg.c	Generated source file that contains Pre-compile configuration data and variables. The file contains content in case of Pre-compile and Post-build configuration variant.	-
CanIf_PBcfg.c	Generated source file that contains Post-build configuration data and variables. The file contains content only in case of Post-build configuration variant.	-

File Name	Description	Integration Tasks
CanIf_CanTrcv.h	Generated header file that contains the includes of the necessary header files of the CAN transceiver drivers used in the system.	-
CanIf_Can.h	Generated header file that contains the include of CanIf_Cbk.h, if a CAN Driver of EQC AR 4.3.1 is used. In all other cases the file is empty.	-

Table 3-1 Implementation files

3.2 Critical Sections

The AUTOSAR standard provides the BSW Scheduler, that handles entering and leaving critical sections. The CAN Interface provides critical section codes that have to be mapped by the BSW Scheduler to following mechanism:

Critical Section Define	Description
CANIF_EXCLUSIVE_AREA_0	Ensures consistency while modifying/changing the CAN-Interface controller mode. > Used inside CanIf_SetControllerMode() > Duration is long > Calls to Can_SetControllerMode() and to several sub-functions
CANIF_EXCLUSIVE_AREA_1	Ensures the consistency of transmit queue. > Used inside CanIf_Init(), CanIf_SetControllerMode(), CanIf_ControllerBusOff(), CanIf_SetPduMode(), CanIf_CancelTxConfirmation() and CanIf_CancelTransmit() > Duration is medium > Calls to several sub-functions
CANIF_EXCLUSIVE_AREA_2	Ensures the consistency of transmit queue and mode handling. > Used inside CanIf_TxConfirmation(), CanIf_CancelTxConfirmation() and CanIf_CancelTxNotification > Duration is long > Calls to Can_Write() and to several sub-functions
CANIF_EXCLUSIVE_AREA_3	Ensures the consistency of transmit queue and mode handling. > Used inside CanIf_SetPduMode() > Duration is medium > Calls to several sub-functions

Critical Section Define	Description
CANIF_EXCLUSIVE_AREA_4	Ensures the consistency while transmission. > Used inside `CanIf_Transmit()` > Duration is long > Calls to `Can_Write()` and to several sub-functions
CANIF_EXCLUSIVE_AREA_5	Ensures the consistency of the dynamic CAN identifier. > Used inside `CanIf_SetDynamicTxId()` > Duration is short > No calls inside
CANIF_EXCLUSIVE_AREA_6	Ensures the consistency of the dynamic J1939 addressing indirection. > Used inside `CanIf_SetAddressTableEntry()` and `CanIf_ResetAddressTableEntry()` > Duration is short > No calls inside
CANIF_EXCLUSIVE_AREA_7	Ensures the consistent usage of the dynamic J1939 addressing informations. > Usage inside `CanIf_RxIndication()` > Duration is short > No calls inside

Table 3-2 Critical Section Codes

If the exclusive areas are entered the upper layer needs to make sure that the CAN interrupts are disabled. Depending on the used CAN Drivers, other mechanisms may have to be used for exclusiv areas with access to the CAN Driver (see respective CAN Driver documentation).

Additionally the following table describes which API of the CAN Interface must not be called during the corresponding area is entered. The CAN Interface API `CanIf_CancelTxNotification() / CanIf_CancelTxConfirmation()` is entered mostly via the CAN interrupt. In case of a platform which confirmation for a transmit cancellation needs to be polled the corresponding API (for example `Can_MainFunction_Write()`) must not be called if the corresponding lock area is entered.

	CANIF_EXCLUSIVE_AREA_0	CANIF_EXCLUSIVE_AREA_1	CANIF_EXCLUSIVE_AREA_2	CANIF_EXCLUSIVE_AREA_3	CANIF_EXCLUSIVE_AREA_4	CANIF_EXCLUSIVE_AREA_5	CANIF_EXCLUSIVE_AREA_6	CANIF_EXCLUSIVE_AREA_7
CanIf_Init	■	■	■	■	■	■	■	■
CanIf_InitMemory	■							
CanIf_Transmit	■	■	■	■	■	■	■	
CanIf_CancelTransmit	■	■	■	■	■			
CanIf_SetControllerMode	■	■	■	■	■			
CanIf_CancelTxNotification/ CanIf_CancelTxConfirmation	■	■	■	■	■			
CanIf_SetPduMode	■	■	■	■	■			
CanIf_TxConfirmation	■	■	■	■	■			
CanIf_ControllerBusOff	■	■	■	■	■			
CanIf_RxIndication							■	
CanIf_SetAddressTableEntry					■		■	■
CanIf_ResetAddressTableEntry					■		■	■

Table 3-3 Restrictions for the different lock areas

3.3 Compiler Abstraction and Memory Mapping

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

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

Compiler Abstraction Definitions / Memory Mapping Sections	CANIF_VAR_ZEROINIT	CANIF_VAR_INIT	CANIF_VAR_NOINIT	CANIF_CONST	CANIF_PBCFG	CANIF_CODE	CANIF_APPL_CODE	CANIF_APPL_VAR	CANIF_APPL_PBCFG	CANIF_VAR_PBCFG
CANIF_START_SEC_CODE CANIF_STOP_SEC_CODE						■				
CANIF_START_SEC_PBCFG CANIF_STOP_SEC_PBCFG					■					
CANIF_START_SEC_CONST_8BIT CANIF_STOP_SEC_CONST_8BIT				■						
CANIF_START_SEC_CONST_32BIT CANIF_STOP_SEC_CONST_32BIT				■						
CANIF_START_SEC_CONST_UNSPECIFIED CANIF_STOP_SEC_CONST_UNSPECIFIED				■						
CANIF_START_SEC_VAR_NOINIT_UNSPECIFIED CANIF_STOP_SEC_VAR_NOINIT_UNSPECIFIED			■							
CANIF_START_SEC_VAR_ZERO_INIT_UNSPECIFIED CANIF_STOP_SEC_VAR_ZERO_INIT_UNSPECIFIED	■									
CANIF_START_SEC_VAR_INIT_UNSPECIFIED CANIF_STOP_SEC_VAR_INIT_UNSPECIFIED		■								
CANIF_START_SEC_VAR_PBCFG CANIF_STOP_SEC_VAR_PBCFG										■

Table 3-4 Compiler abstraction and memory mapping

The Compiler Abstraction Definitions `CANIF_APPL_CODE`, `CANIF_APPL_VAR` and `CANIF_APPL_PBCFG` are used to address code, variables and constants which are declared by other modules and used by the CAN Interface.

These definitions are not mapped by the CAN Interface but by the memory mapping realized in the CAN Driver, CAN Transceiver Driver, PDU Router, Network management, Transport Protocol Layer, ECU State Manager and the CAN State manager.

The CAN Driver definition(s) `CAN_<VendorId>_<VendorApiInfix>_APPL_VAR` and the CAN Interface definition `CANIF_APPL_VAR` must be mapped to the same memory.

3.4 Can_GeneralTypes

The `Can_GeneralTypes.h` is provided as a template file as part of the delivery. If the CAN Stack contains only MICROSAR Classic components no further integrations steps are required. If at least one third party CAN Driver is used, please follow the advice in the technical reference of the “3rdParty MCAL Integration” package.

Note
If the configuration contains a CAN Driver of EQC AR 4.3.1 `uint32` is assumed by the CAN Interface for the data type `Can_IdType`.

If the configuration contains at least one third party CAN Driver `uint16` is assumed by the CAN Interface for the data type `Can_HwHandleType`.

4 API Description

For an interfaces overview please see Figure 1-1.

4.1 Services provided by CAN Interface

4.1.1 CanIf_InitMemory

Prototype	
`void CanIf_InitMemory ( void )`	
Parameter	
-	-
Return code	
-	-
Functional Description	
Initializes global RAM variables, which have to be available before any other API of the CAN-Interface is called. Sets the CAN-Interface to the state: uninitialized.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is non-reentrant. > May only be called once before `CanIf_Init()`. > Configuration Variant(s): -	
Call context	
> TASK	

Table 4-1 CanIf_InitMemory

4.1.2 CanIf_Init

Prototype	
`void CanIf_Init ( const CanIf_ConfigType *ConfigPtr )`	
Parameter	
`ConfigPtr [in]`	Pointer to the CanIf_Config structure. If multiple configurations are available, the active configuration can be selected by using the related CanIf_Config_<IdentityName> structure.
Return code	
-	-
Functional Description	
Initializes the CAN-Interface.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is non-reentrant. > The function `CanIf_InitMemory()` must be called before the function `CanIf_Init()` is called. This function must be called before any other service functionality of the CAN-Interface. > Configuration Variant(s): -	
Call context	
> TASK	

Table 4-2 Canif_Init

4.1.3 CanIf_SetControllerMode

Prototype	
`Std_ReturnType` **`CanIf_SetControllerMode`** `( uint8 ControllerId, CanIf_ControllerModeType ControllerMode )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf ControllerId which is assigned to a CAN controller, which is requested for controller mode transition.
`ControllerMode [in]`	Requested controller mode transition.
Return code	
`E_OK`	The request to change the controller mode has been accepted.
`E_NOT_OK`	The request to change the controller mode has not been accepted.
Functional Description	

Requests a controller mode transition of a CAN controller.
Supported controller modes:
`CANIF_CS_SLEEP`
`CANIF_CS_STOPPED`
`CANIF_CS_STARTED`	

Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is asynchronous. > This function is non-reentrant. > Possible preconditions (e.g. interrupt lock) for calling Can_SetControllerMode can be found in the respective CAN Driver documentation. > Configuration Variant(s): -	
Call context	
> ANY	

Table 4-3 CanIf_SetControllerMode

4.1.4 CanIf_GetControllerMode

Prototype	
`Std_ReturnType` **`CanIf_GetControllerMode`** `( uint8 ControllerId, CanIf_ControllerModeType *ControllerModePtr )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf ControllerId which is assigned to a CAN controller, which is requested for its current controller mode.
`ControllerModePtr [out]`	Pointer to memory location, where the current controller mode of the CAN controller will be stored.
Return code	
`E_OK`	Controller mode request has been accepted; current controller mode is stored at `ControllerModePtr`.
`E_NOT_OK`	Controller mode request has not been accepted.
Functional Description	
Returns the current controller mode of a CAN controller.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is reentrant. > Configuration Variant(s): -	
Call context	
> ANY	

Table 4-4 CanIf_GetControllerMode

4.1.5 CanIf_Transmit

Prototype	
`Std_ReturnType` **`CanIf_Transmit`** `( PduIdType CanTxPduId, const PduInfoType *PduInfoPtr )`	
Parameter	
`CanTxPduId [in]`	Handle of Tx-PDU which shall be transmitted.
`PduInfoPtr [in]`	Pointer to a struct containing the properties of the Tx PDU.
Return codine	
`E_OK`	Transmit request has been accepted.
`E_NOT_OK`	Transmit request has not been accepted.
Functional Description	
Initiates a request for transmission of the specified Tx-PDU.	
Particularities and Limitations	
> Service ID: see table 'Service IDs'	
> This function is synchronous.	
> This function is reentrant (only for a different `CanTxPduId`).	
> Configuration Variant(s): -	
Call context	
> ANY	

Table 4-5 CanIf_Transmit

4.1.6 CanIf_CancelTransmit

Prototype	
`Std_ReturnType` **`CanIf_CancelTransmit`** `( PduIdType CanTxPduId )`	
Parameter	
`CanTxPduId [in]`	Handle of Tx-PDU which shall be canceled.
Return code	
`E_OK`	Transmit cancellation request has been accepted.
`E_NOT_OK`	Transmit cancellation request has not been accepted.
Functional Description	
Initiates the cancellation / suppression of the confirmation of the specified Tx-PDU.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is non-reentrant. > AUTOSAR only defines a dummy function. For MICROSAR Classic this function has the functionality to cancel an ordered Tx-PDU. > Configuration Variant(s): `CANIF_CANCEL_SUPPORT_API == STD_ON`	
Call context	
> ANY	

Table 4-6 CanIf_CancelTransmit

4.1.7 CanIf_SetPduMode

Prototype	
`Std_ReturnType CanIf_SetPduMode ( uint8 ControllerId, CanIf_PduSetModeType PduModeRequest )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf ControllerId which is assigned to a CAN controller, which is requested for PDU mode transition.
`PduModeRequest [in]`	Requested PDU mode transition.
Return code	
`E_OK`	The request to change the PDU mode has been accepted.
`E_NOT_OK`	The request to change the PDU mode has not been accepted.
Functional Description	

Requests a PDU mode transition of a CAN controller.
Supported PDU modes:
`CANIF_SET_OFFLINE`
`CANIF_SET_RX_OFFLINE`
`CANIF_SET_RX_ONLINE`
`CANIF_SET_TX_OFFLINE`
`CANIF_SET_TX_ONLINE`
`CANIF_SET_ONLINE`
`CANIF_SET_TX_OFFLINE_ACTIVE`	

Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is non-reentrant. > Controller has to be in state `CANIF_CS_STARTED`. > Configuration Variant(s): -	
Call context	
> ANY	

Table 4-7 CanIf_SetPduMode

4.1.8 CanIf_GetPduMode

Prototype	
`Std_ReturnType` **`CanIf_GetPduMode`** `( uint8 ControllerId, CanIf_PduGetModeType *PduModePtr )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf ControllerId which is assigned to a CAN controller, which is requested for its current PDU mode.
`PduModePtr [out]`	Pointer to memory location, where the current PDU mode of the CAN controller will be stored.
Return codein	
`E_OK`	PDU mode request has been accepted; current PDU mode is stored at `PduModePtr`.
`E_NOT_OK`	PDU mode request has not been accepted.
Functional Description	
Returns the current PDU mode of a CAN controller.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is reentrant. > Configuration Variant(s): -	
Call context	
> ANY	

Table 4-8 CanIf_GetPduMode

4.1.9 CanIf_GetVersionInfo

Prototype	
`void` **`CanIf_GetVersionInfo`** `( Std_VersionInfoType *VersionInfo )`	
Parameter	
`VersionInfo [out]`	Pointer to variable Pointer to memory location, where the version information of this module will be stored.
Return code	
-	-
Functional Description	
Returns the version information of the called CAN Interface module. Version information (BCD-coded): Vendor ID AUTOSAR module ID SW version of the component	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is reentrant > Configuration Variant(s): `CANIF_VERSION_INFO_API == STD_ON`	
Call context	
> ANY	

Table 4-9 CanIf_GetVersionInfo

4.1.10 CanIf_SetDynamicTxId

Prototype	
`void` **`CanIf_SetDynamicTxId`** `( PduIdType CanTxPduId, Can_IdType CanId )`	
Parameter	
`CanTxPduId [in]`	Handle of Tx-PDU which CAN identifier shall be modified.
`CanId [in]`	CAN identifier which shall be set for the specified Tx-PDU.
Return code	
-	-
Functional Description	
Reconfigures the CAN identifier of the specified Tx-PDU.	
Particularities and Limitations	
> Service ID: see table 'Service IDs'	
> This function is synchronous.	
> This function is reentrant.	
> Shall not be interrupted by a call of `CanIf_Transmit()` for the same Tx PDU.	
> Configuration Variant(s): `CANIF_SETDYNAMICTXID_API == STD_ON`	
Call context	
> ANY	

Table 4-10 CanIf_SetDynamicTxId

4.1.11 CanIf_SetTrcvMode

Prototype	
`Std_ReturnType` **`CanIf_SetTrcvMode`** `( uint8 TransceiverId, CanTrcv_TrcvModeType TransceiverMode )`	
Parameter	
`TransceiverId [in]`	Abstracted CanIf TransceiverId, which is assigned to a CAN transceiver, which is requested for mode transition.
`TransceiverMode [in]`	Requested transceiver mode transition.
Return code	
`E_OK`	Transceiver mode request has been accepted.
`E_NOT_OK`	Transceiver mode request has not been accepted.
Functional Description	

Requests a transceiver mode transition of a CAN transceiver.
Supported transceiver modes:
`CANTRCV_TRCVMODE_NORMAL`
`CANTRCV_TRCVMODE_SLEEP`
`CANTRCV_TRCVMODE_STANDBY`	

Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous for each CAN transceiver, which is configured synchronous else asynchronous. > This function is non-reentrant > Configuration Variant(s): `CANIF_TRCV_HANDLING == STD_ON`	
Call context	
> TASK	

Table 4-11 CanIf_SetTrcvMode

4.1.12 CanIf_GetTrcvMode

Prototype	
`Std_ReturnType` **`CanIf_GetTrcvMode`** `( CanTrcv_TrcvModeType *TransceiverModePtr, uint8 TransceiverId )`	
Parameter	
`TransceiverId [in]`	Abstracted CanIf TransceiverId, which is assigned to a CAN transceiver, which is requested for current transceiver mode.
`TransceiverModePtr [out]`	Pointer to memory location, where the current transceiver mode of the CAN transceiver will be stored.
Return code	
`E_OK`	Transceiver mode request has been accepted; current transceiver mode is stored at `TransceiverModePtr`.
`E_NOT_OK`	Transceiver mode request has not been accepted.
Functional Description	
Returns the current transceiver mode of a CAN transceiver.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous > This function is non-reentrant > Configuration Variant(s): `CANIF_TRCV_HANDLING == STD_ON`	
Call context	
> TASK	

Table 4-12 CanIf_GetTrcvMode

4.1.13 CanIf_GetTrcvWakeupReason

Prototype	
`Std_ReturnType` **`CanIf_GetTrcvWakeupReason`** `( uint8 TransceiverId, CanTrcv_TrcvWakeupReasonType *TrcvWuReasonPtr )`	
Parameter	
`TransceiverId [in]`	Abstracted CanIf TransceiverId, which is assigned to a CAN transceiver, which is requested for wakeup reason.
`TrcvWuReasonPtr [out]`	Pointer to memory location, where the wake-up reason of the CAN transceiver will be stored.
Return code	
`E_OK`	Transceiver wakeup reason request has been accepted; wakeup reason is stored at `TrcvWuReasonPtr`.
`E_NOT_OK`	Transceiver wakeup reason request has not been accepted.
Functional Description	
Returns the wakeup reason of a CAN transceiver.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is non-reentrant. > Configuration Variant(s): `CANIF_TRCV_HANDLING == STD_ON`	
Call context	
> ANY	

Table 4-13 CanIf_GetTrcvWakeupReason

4.1.14 CanIf_SetTrcvWakeupMode

Prototype	
`Std_ReturnType` **`CanIf_SetTrcvWakeupMode`** `( uint8 TransceiverId, CanTrcv_TrcvWakeupModeType TrcvWakeupMode )`	
Parameter	
`TransceiverId [in]`	Abstracted CanIf TransceiverId, which is assigned to a CAN transceiver, which is requested for wakeup mode transition.
`TrcvWakeupMode [in]`	Requested transceiver wakeup mode transition.
Return code	
`E_OK`	Transceiver wakeup mode request has been accepted.
`E_NOT_OK`	Transceiver wakeup mode request has not been accepted.
Functional Description	

Requests a transceiver wakeup mode transition of a CAN transceiver.
Supported wakeup modes:
`CANTRCV_WUMODE_ENABLE`
`CANTRCV_WUMODE_DISABLE`
`CANTRCV_WUMODE_CLEAR`	

Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is non-reentrant. > Configuration Variant(s): `CANIF_TRCV_HANDLING == STD_ON`	
Call context	
> ANY	

Table 4-14 CanIf_SetTrcvWakeupMode

4.1.15 CanIf_CheckWakeup

Prototype	
`Std_ReturnType` **`CanIf_CheckWakeup`** `( EcuM_WakeupSourceType WakeupSource )`	
Parameter	
`WakeupSource [in]`	Source device, which initiated the wakeup event (CAN controller or CAN transceiver).
Return code	
`E_OK`	The specified source device signals the wakeup event.
`E_NOT_OK`	Check wakeup request has not been accepted.
Functional Description	
Checks, whether an underlying CAN controller or CAN transceiver signals a wakeup event.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is reentrant. > Configuration Variant(s): `CANIF_WAKEUP_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-15 CanIf_CheckWakeup

4.1.16 CanIf_CheckValidation

Prototype	
`Std_ReturnType` **`CanIf_CheckValidation`** `( EcuM_WakeupSourceType WakeupSource )`	
Parameter	
`WakeupSource [in]`	Source device which initiated the wakeup event and which has to be validated (CAN controller or CAN transceiver).
Return code	
`E_OK`	Check validation request has been accepted.
`E_NOT_OK`	Check validation request has not been accepted.
Functional Description	
Checks if a Rx PDU was received since the last wake-up event. If a Rx PDU was received between the call of `CanIf_CheckWakeup` and `CanIf_CheckValidation` the configurable EcuM callback function (default name "`EcuM_ValidateWakeupEvent`") is called from the context of this function.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is reentrant. > `CanIf_CheckWakeup` has to be called before and a wakeup event has to be detected. > CAN Interface has to be set to `CANIF_CS_STARTED` mode before a validation is possible. > Configuration Variant(s): `CANIF_WAKEUP_SUPPORT == STD_ON` and `CANIF_WAKEUP_VALIDATION == STD_ON`	
Call context	
> ANY	

Table 4-16 CanIf_CheckValidation

4.1.17 CanIf_GetTxConfirmationState

Prototype	
`CanIf_NotifStatusType` **`CanIf_GetTxConfirmationState`** `( uint8 ControllerId )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf ControllerId which is assigned to a CAN controller, which is requested for its Tx confirmation state.
Return code	
`CANIF_NO_NOTIFICATION`	No transmit event occurred for the requested CAN controller.
`CANIF_TX_RX_NOTIFICATION`	The requested CAN controller has successfully transmitted any message.
Functional Description	
Reports if any TX confirmation has been done for the whole CAN controller since it's last start.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is reentrant. > Configuration Variant(s): `CANIF_PUBLIC_TX_CONFIRM_POLLING_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-17 CanIf_GetTxConfirmationState

4.1.18 CanIf_ClearTrcvWufFlag

Prototype	
`Std_ReturnType` **`CanIf_ClearTrcvWufFlag`** `( uint8 TransceiverId )`	
Parameter	
`TransceiverId [in]`	Abstracted CanIf TransceiverId, which is assigned to a CAN transceiver, which is requested to clear its WUF flag.
Return code	
`E_OK`	Clear WUF flag request has been accepted.
`E_NOT_OK`	Clear WUF flag request has not been accepted.
Functional Description	
Requests a CAN transceiver to clear its WUF flag.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous for each CAN transceiver which is configured synchronous else asynchronous. > This function is reentrant. > Configuration Variant(s): `CANIF_TRCV_HANDLING == STD_ON` and `CANIF_PN_TRCV_HANDLING == STD_ON`	
Call context	
> ANY	

Table 4-18 CanIf_ClearTrcvWufFlag

4.1.19 CanIf_CheckTrcvWakeFlag

Prototype	
`Std_ReturnType` **`CanIf_CheckTrcvWakeFlag`** `( uint8 TransceiverId )`	
Parameter	
`TransceiverId [in]`	Abstracted CanIf TransceiverId, which is assigned to a CAN transceiver, which is requested to check its Wake flag.
Return code	
`E_OK`	Check Wake flag request has been accepted.
`E_NOT_OK`	Check Wake flag request has not been accepted.
Functional Description	
Requests a CAN transceiver to check for Wake flag.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous for each CAN transceiver which is configured synchronous else asynchronous. > This function is reentrant. > Configuration Variant(s): `CANIF_TRCV_HANDLING == STD_ON` and `CANIF_PN_TRCV_HANDLING == STD_ON`	
Call context	
> ANY	

Table 4-19 CanIf_CheckTrcvWakeFlag

4.1.20 CanIf_SetBaudrate

Prototype	
`Std_ReturnType` **`CanIf_SetBaudrate`** `( uint8 ControllerId, uint16 BaudRateConfigID )`	
Parameter	
`ControllerId [in]`	Abstract CanIf ControllerId which is assigned to a CAN controller, whose baudrate shall be set.
`BaudRateConfigID [in]`	References a baudrate configuration by ID.
Return code	
`E_OK`	Set baudrate request has been accepted. Baudrate change started.
`E_NOT_OK`	Set baudrate request has not been accepted.
Functional Description	
Requests to set the baudrate configuration of the specified CAN controller.	
Particularities and Limitations	
> Service ID: see table 'Service IDs'	
> This function is synchronous.	
> This function is non-reentrant	
> Configuration Variant(s): `CANIF_SET_BAUDRATE_API == STD_ON`	
Call context	
> ANY	

Table 4-20 CanIf_SetBaudrate

4.1.21 CanIf_ChangeBaudrate

Prototype	
`Std_ReturnType` **`CanIf_ChangeBaudrate`** `( uint8 ControllerId, const uint16 Baudrate )`	
Parameter	
`ControllerId [in]`	Abstract CanIf ControllerId which is assigned to a CAN controller, whose baudrate shall be set.
`Baudrate [in]`	Requested baudrate [kbps].
Return code	
`E_OK`	Change baudrate request has been accepted. Baudrate change started.
`E_NOT_OK`	Change baudrate request has not been accepted.
Functional Description	
Requests to change the baudrate of the specified CAN controller. If possible please use the API `CanIf_SetBaudrate()` instead of this one.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is non-reentrant. > Configuration Variant(s): `CANIF_CHANGE_BAUDRATE_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-21 CanIf_ChangeBaudrate

4.1.22 CanIf_CheckBaudrate

Prototype	
`Std_ReturnType` **`CanIf_CheckBaudrate`** `( uint8 ControllerId, const uint16 Baudrate )`	
Parameter	
`ControllerId [in]`	Abstract CanIf ControllerId which is assigned to a CAN controller, whose baudrate support shall be checked.
`Baudrate [in]`	Baudrate to check [kbps].
Return code	
`E_OK`	Baudrate is supported by the CAN controller.
`E_NOT_OK`	Baudrate is not supported / invalid by the CAN controller.
Functional Description	
Checks if a CAN controller supports the specified baudrate.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is non-reentrant. > Configuration Variant(s): `CANIF_CHANGE_BAUDRATE_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-22 CanIf_CheckBaudrate

4.1.23 CanIf_SetAddressTableEntry

Prototype	
`void CanIf_SetAddressTableEntry ( uint8 ControllerId, uint8 intAddr, uint8 busAddr )`	
Parameter	
`ControllerId [in]`	Abstract CanIf ControllerId which is assigned to a CAN controller, at which a J1939 address shall be set.
`intAddr [in]`	J1939 internal address.
`busAddr [in]`	J1939 bus address.
Return code	
-	-
Functional Description	
Sets up one relation between internal and external address. Only used in J1939 environment.	
Particularities and Limitations	
> Service ID: see table 'Service IDs'	
> This function is synchronous.	
> This function is non-reentrant.	
> Configuration Variant(s): `CANIF_J1939_DYN_ADDR_SUPPORT != CANIF_J1939_DYN_ADDR_DISABLED`	
Call context	
> ANY	

Table 4-23 CanIf_SetAddressTableEntry

4.1.24 CanIf_ResetAddressTableEntry

Prototype	
`void CanIf_ResetAddressTableEntry ( uint8 ControllerId, uint8 intAddr )`	
Parameter	
`ControllerId [in]`	Abstract CanIf ControllerId which is assigned to a CAN controller, at which a J1939 address shall be reset.
`intAddr [in]`	J1939 internal address.
Return code	
-	-
Functional Description	
Resets one relation between internal and external address. Only used in J1939 environment.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is non-reentrant. > Configuration Variant(s): `CANIF_J1939_DYN_ADDR_SUPPORT != CANIF_J1939_DYN_ADDR_DISABLED`	
Call context	
> ANY	

Table 4-24 CanIf_ResetAddressTableEntry

4.1.25 CanIf_GetTxAddressTableEntry

Prototype	
`uint8 CanIf_GetTxAddressTableEntry ( uint8 ControllerId, uint8 address )`	
Parameter	
`ControllerId [in]`	Abstract CanIf ControllerId which is assigned to a CAN Controller.
`address [in]`	Used to retrieve table entry of Tx lookup table.
Return code	
0-255	Table entry value (input parameter "address" is returned if table entry value can not be determined).
Functional Description	
Returns the entry at the Tx lookup table position corresponding to the passed CAN Controller and address.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is reentrant. > Configuration Variant(s): `CANIF_J1939_DYN_ADDR_SUPPORT != CANIF_J1939_DYN_ADDR_DISABLED`	
Call context	
> ANY	

Table 4-25 CanIf_GetTxAddressTableEntry

4.1.26 CanIf_GetRxAddressTableEntry

Prototype	
`uint8 CanIf_GetRxAddressTableEntry ( uint8 ControllerId, uint8 address )`	
Parameter	
`ControllerId [in]`	Abstract CanIf ControllerId which is assigned to a CAN Controller.
`address [in]`	Used to retrieve table entry of Rx lookup table.
Return code	
0-255	Table entry value (input parameter "address" is returned if table entry value can not be determined).
Functional Description	
Returns the entry at the Rx lookup table position corresponding to the passed CAN Controller and address.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is reentrant. > Configuration Variant(s): `CANIF_J1939_DYN_ADDR_SUPPORT != CANIF_J1939_DYN_ADDR_DISABLED`	
Call context	
> ANY	

Table 4-26 CanIf_GetRxAddressTableEntry

4.1.27 CanIf_RamCheckExecute

Prototype	
`void CanIf_RamCheckExecute ( uint8 ControllerId )`	
Parameter	
`ControllerId [in]`	Abstract CanIf ControllerId which is assigned to a CAN controller, for which the RAM-check shall be executed.
Return code	
–	–
Functional Description	
Requests the specified CAN controller to execute the RAM check of its CAN controller HW-registers.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is non-reentrant. > Configuration Variant(s): `CANIF_EXTENDED_RAM_CHECK_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-27 CanIf_RamCheckExecute

4.1.28 CanIf_RamCheckEnableMailbox

Prototype	
`void CanIf_RamCheckEnableMailbox ( uint8 ControllerId, CanIf_HwHandleType HwHandle )`	
Parameter	
`ControllerId [in]`	Abstract CanIf ControllerId which is assigned to a CAN controller, for which the mailbox shall be enabled.
`HwHandle [in]`	Handle of the mailbox, which shall be enabled.
Return code	
–	–
Functional Description	
Reactivates the specified mailbox from a CAN controller after it was deactivated by RAM check.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is non-reentrant. > Configuration Variant(s): `CANIF_EXTENDED_RAM_CHECK_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-28 CanIf_RamCheckEnableMailbox

4.1.29 CanIf_RamCheckEnableController

Prototype	
`void CanIf_RamCheckEnableController ( uint8 ControllerId )`	
Parameter	
`ControllerId [in]`	Abstract CanIf ControllerId which is assigned to a CAN controller, which shall be enabled.
Return code	
-	-
Functional Description	
Reactivates a CAN controller after it was deactivated by RAM check.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is non-reentrant. > Configuration Variant(s): `CANIF_EXTENDED_RAM_CHECK_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-29 CanIf_RamCheckEnableController

4.1.30 CanIf_SetPduReceptionMode

Prototype	
`Std_ReturnType` **`CanIf_SetPduReceptionMode`** `( PduIdType id, CanIf_ReceptionModeType mode )`	
Parameter	
`id [in]`	The handle of Rx-PDU whose reception mode shall be changed.
`mode [in]`	The reception mode which shall be set. Following reception modes are possible: > `CANIF_RMT_IGNORE_CONTINUE`: In case of a match the received Rx-PDU is not forwarded to configured upper layer and the search for a potential match continues. > `CANIF_RMT_RECEIVE_STOP`: In case of a match the received Rx-PDU is forwarded to configured upper layer.
Return code	
`E_OK`	Set PDU reception mode request has been accepted. PDU reception mode was changed.
`E_NOT_OK`	Set PDU reception mode request has not been accepted. PDU reception mode was not changed.
Functional Description	
Sets the reception mode of a Rx-PDU. With this API the upper layer may influence on which PDU ID a CAN ID is received, respective if a CAN ID is received at all. During the initialization the reception mode of all affected Rx-PDUs is set to `CANIF_RMT_RECEIVE_STOP`.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is reentrant for different Rx-PDU handles. > Configuration Variant(s): `CANIF_SET_PDU_RECEPTION_MODE_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-30 CanIf_SetPduReceptionMode

4.1.31 CanIf_GetControllerErrorState

Prototype	
`Std_ReturnType` **`CanIf_GetControllerErrorState`** `( uint8 ControllerId, Can_ErrorStateType *ErrorStatePtr )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf ControllerId which is assigned to a CAN controller, from which the error state is requested.
`ErrorStatePtr [out]`	Pointer to a memory location, where the error state of the CAN controller will be stored.
Return code	
`E_OK`	Error state request has been accepted; current error state is stored at `ErrorStatePtr`.
`E_NOT_OK`	Error state request has not been accepted.
Functional Description	
Returns the current error state of a CAN controller.	
Particularities and Limitations	
> Service ID: see table 'Service IDs'	
> This function is synchronous.	
> This function is reentrant (only for different `ControllerId`).	
> Configuration Variant(s): `CANIF_BUS_MIRRORING_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-31 CanIf_GetControllerErrorState

4.1.32 CanIf_GetControllerRxErrorCounter

Prototype	
`Std_ReturnType` **`CanIf_GetControllerRxErrorCounter`** `( uint8 ControllerId, uint8 *RxErrorCounterPtr )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf ControllerId which is assigned to a CAN controller, from which the Rx error counter is requested.
`RxErrorCounterPtr [out]`	Pointer to a memory location, where the current Rx error counter of the CAN controller will be stored.
Return code	
`E_OK`	Rx error counter request has been accepted; current Rx error counter is stored at `RxErrorCounterPtr`.
`E_NOT_OK`	Rx error counter request has not been accepted.
Functional Description	
Returns the current Rx error counter of a CAN controller.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is reentrant (only for different `ControllerId`). > Configuration Variant(s): `CANIF_BUS_MIRRORING_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-32 CanIf_GetControllerRxErrorCounter

4.1.33 CanIf_GetControllerTxErrorCounter

Prototype	
`Std_ReturnType` **`CanIf_GetControllerTxErrorCounter`** `( uint8 ControllerId, uint8 *TxErrorCounterPtr )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf ControllerId which is assigned to a CAN controller, from which the Tx error counter is requested.
`TxErrorCounterPtr [out]`	Pointer to a memory location, where the current Tx error counter of the CAN controller will be stored.
Return code	
`E_OK`	Tx error counter request has been accepted; current Tx error counter is stored at `TxErrorCounterPtr`.
`E_NOT_OK`	Tx error counter request has not been accepted.
Functional Description	
Returns the current Tx error counter of a CAN controller.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is reentrant (only for different `ControllerId`). > Configuration Variant(s): `CANIF_BUS_MIRRORING_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-33 CanIf_GetControllerTxErrorCounter

4.1.34 CanIf_EnableBusMirroring

Prototype	
`Std_ReturnType` **`CanIf_EnableBusMirroring`** `( uint8 ControllerId, boolean MirroringActive )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf ControllerId which is assigned to a CAN controller, for which mirroring shall be enabled or disabled.
`MirroringActive [in]`	`TRUE`: Mirroring will be enabled to report each successful received or transmitted CAN frame on the given CAN controller to the Mirror module. `FALSE`: Mirroring will be disabled to not report received or transmitted CAN frame on the given CAN controller to the Mirror module.
Return code	
`E_OK`	Mirroring change request has been accepted; mirroring was changed for the given CAN controller.
`E_NOT_OK`	Mirroring change request has not been accepted; mirroring was not changed for the given CAN controller.
Functional Description	
Enables or disables mirroring for a CAN controller. During the initialization mirroring of all configured CAN controller is set to disabled.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is reentrant. > Configuration Variant(s): `CANIF_BUS_MIRRORING_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-34 CanIf_EnableBusMirroring

4.2 Services used by CAN Interface

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

Component	API
Appl	- User_RxIndication (*) - User_TxConfirmation (*)
Can	- Can_SetControllerMode - Can_Write - CanXL_Write - Can_CancelTx - Can_CheckWakeup - Can_CheckBaudrate - Can_ChangeBaudrate - Can_SetBaudrate - Can_RamCheckExecute - Can_RamCheckEnableMailbox - Can_RamCheckEnableController - Can_GetControllerErrorState - Can_GetControllerRxErrorCounter - Can_GetControllerTxErrorCounter
CanNm	- CanNm_RxIndication - CanNm_TxConfirmation
CanSm	- CanSM_CheckTransceiverWakeFlagIndication - CanSM_ClearTrcvWufFlagIndication - CanSM_ConfirmPnAvailability - CanSM_ControllerBusOff - CanSM_ControllerModeIndication - CanSM_TransceiverModeIndication - CanSM_RamCheckCorruptController - CanSM_RamCheckCorruptMailbox
CanTp	- CanTp_RxIndication - CanTp_TxConfirmation
CanTSyn	- CanTSyn_RxIndication - CanTSyn_TxConfirmation
CanTrcv	- CanTrcv_SetOpMode - CanTrcv_GetOpMode - CanTrcv_SetWakeupMode - CanTrcv_CheckWakeup - CanTrcv_GetBusWuReason
CCP	- Ccp_RxIndication - Ccp_TxConfirmation

Component	API
CDD	- My_DataChecksumRxErrFct (*) - User_CheckTransceiverWakeFlagIndication (*) - User_ClearTrcvWufFlagIndication (*) - User_ConfirmPnAvailability (*) - User_ControllerBusOff (*) - User_ControllerModeIndication (*) - User_RamCheckCorruptController (*) - User_RamCheckCorruptMailbox (*) - User_TransceiverModeIndication (*) - User_ValidateWakeupEvent (*) - User_RxIndication (*) - User_TxConfirmation (*)
Det	- Det_ReportError
EcuM	- EcuM_ValidateWakeupEvent - EcuM_BswErrorHook
IdsM	- IdsM_SetSecurityEventWithContextData
J1939Nm	- J1939Nm_RxIndication - J1939Nm_TxConfirmation
J1939Tp	- J1939Tp_RxIndication - J1939Tp_TxConfirmation
Mirror	- Mirror_ReportCanFrame
PduR	- PduR_CanIfRxIndication - PduR_CanIfTxConfirmation
SchM	- SchM_Enter_CanIf_<ExclusiveArea> - SchM_Exit_CanIf_<ExclusiveArea>
VStdLib	- VStdLib_MemCpy - VStdLib_GetHighestBitPosOne8 - VStdLib_GetHighestBitPosOne16 - VStdLib_GetHighestBitPosOne32 - VStdLib_GetHighestBitPosOne64
Xcp	- Xcp_CanIfRxIndication - Xcp_CanIfTxConfirmation

Table 4-35 Services used by the CAN Interface

* Names of the call back functions can be configured freely.

4.3 Callback Functions

This chapter describes the callback functions that are implemented by the CAN Interface and can be invoked by other modules. The prototypes of the callback functions are provided in the header files `CanIf_Cbk.h`, `CanIf.h` and `CanIf_Cfg.h` by the CAN Interface.

4.3.1 CanIf_TxConfirmation

Prototype	
`void CanIf_TxConfirmation ( PduIdType CanTxPduId )`	
Parameter	
`CanTxPduId [in]`	Handle of the Tx-PDU which is successfully transmitted.
Return code	
-	-
Functional Description	
Called by the CAN Driver to notify the CAN Interface about the successful transmission of the specified Tx-PDU.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is reentrant. > Configuration Variant(s): -	
Call context	
> ANY	

Table 4-36 CanIf_TxConfirmation

4.3.2 CanIf_RxIndication

Prototype	
`void CanIf_RxIndication (const Can_HwType* Mailbox, const PduInfoType* PduInfoPtr)`	
Parameter	
`Mailbox [in]`	Hardware handle where the Rx PDU was received in and its corresponding CAN controller.
`PduInfoPtr [in]`	Pointer to the received Rx PDU.
Return code	
-	-
Functional Description	
Called by the CAN Driver of EQC AR 4.2.1 or AR 4.3.1 to notify the CAN Interface about the reception of the specified CAN 2.0/FD Rx PDU. This function searches whether the received PDU matches one of the configured ones. If yes, then the configured upper layer is notified.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is reentrant. > Contained in CanIf_Cfg.h, if a CAN Driver of EQC AR 4.2.1 or AR 4.3.1 is configured. > Configuration Variant(s): -	
Call context	
> ANY	

Table 4-37 CanIf_RxIndication

4.3.3 CanIf_RxIndication (CAN Driver of EQC AR 4.0.3 or MSRC)

Prototype	
`void CanIf_RxIndication ( CanIf_HwHandleType Hrh, Can_IdType CanId, uint8 CanDlc, const uint8 *CanSduPtr )`	
Parameter	
`Hrh [in]`	Hardware handle where the Rx PDU was received in.
`CanId [in]`	CAN identifier of the received Rx PDU.
`CanDlc [in]`	Data length of the received Rx PDU.
`CanSduPtr [in]`	Pointer to the data of the received Rx PDU.
Return code	
-	-
Functional Description	
Called by the CAN Driver of EQC AR 4.0.3 or MSRC to notify the CAN Interface about the reception of the specified CAN 2.0/FD Rx PDU. This function searches whether the received PDU matches one of the configured ones. If yes, then the configured upper layer is notified.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is reentrant. > Contained in CanIf_Cfg.h, if a CAN Driver of EQC AR 4.0.3 or MSRC is configured. > Configuration Variant(s): -	
Call context	
> ANY	

Table 4-38 CanIf_RxIndication (CAN Driver of EQC AR 4.0.3 or MSRC)

4.3.4 CanIf_XLRxIndication

Prototype	
`void CanIf_XLRxIndication (const CanXL_HwType* Mailbox, const PduInfoType* PduInfoPtr)`	
Parameter	
`Mailbox [in]`	Contains the CAN XL Hrh, the abstracted CanIf ControllerId (belonging to the CAN controller at which the CAN XL Rx PDU was received) and the CAN XL parameters..
`PduInfoPtr [in]`	Pointer to the received CAN XL Rx PDU.
Return code	
-	-
Functional Description	
Called by the CAN Driver to notify the CAN Interface about the reception of the specified CAN XL Rx PDU.This function searches whether the received CAN XL Rx PDU matches one of the configured ones. If yes, then the configured upper layer is notified.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is reentrant. > Configuration Variant(s): `CANIF_SUPPORT_CAN_XL == STD_ON`	
Call context	
> ANY	

Table 4-39 CanIf_XLRxIndication

4.3.5 CanIf_CancelTxConfirmation

Prototype	
`void CanIf_CancelTxConfirmation ( PduIdType CanTxPduId, const Can_PduType *PduInfoPtr )`	
Parameter	
`CanTxPduId [in]`	Handle of the Tx-PDU which was cancelled.
`PduInfoPtr [in]`	Pointer to parameters of the canceled Tx-PDU.
Return code	
-	-
Functional Description	
Called by the CAN Driver to notify the CAN Interface about the transmission cancellation of the specifed Tx-PDU. The specified Tx-PDU is re-queued in a transmit-buffer.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is reentrant. > Configuration Variant(s): `CANIF_TRANSMIT_BUFFER == STD_ON` and `CANIF_TRANSMIT_CANCELLATION == STD_ON`	
Call context	
> ANY	

Table 4-40 CanIf_CancelTxConfirmation

4.3.6 CanIf_ControllerBusOff

Prototype	
`void CanIf_ControllerBusOff ( uint8 ControllerId )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf ControllerId which is assigned to a CAN controller, at which the BusOff-event occurred.
Return code	
-	-
Functional Description	
Called by the CAN Driver to notify the CAN Interface about an occurred BusOff-event at the specified CAN controller.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is reentrant. > Configuration Variant(s): -	
Call context	
> ANY	

Table 4-41 CanIf_ControllerBusOff

4.3.7 CanIf_CancelTxNotification

Prototype	
`void CanIf_CancelTxNotification ( PduIdType PduId, CanIf_CancelResultType IsCancelled )`	
Parameter	
`PduId [in]`	Handle of the Tx-PDU which was cancelled.
`IsCancelled [in]`	Parameter currently not evaluated.
Return code	
-	-
Functional Description	
Called by the CAN Driver to notify the CAN Interface about the transmission cancelation of the specified Tx-PDU. The specified Tx-PDU is not confirmed to the configured upper layer. The next Tx-PDU from the transmit-buffer is transmitted. Used for trigger-purpose to fill the free HW-object, after calling of `CanIf_CancelTransmit()`.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is reentrant. > Configuration Variant(s): `CANIF_CANCEL_SUPPORT_API == STD_ON` > None AUTOSAR API	
Call context	
> ANY	

Table 4-42 CanIf_CancelTxNotification

4.3.8 CanIf_TrcvModeIndication

Prototype	
`void CanIf_TrcvModeIndication ( uint8 TransceiverId, CanTrcv_TrcvModeType TransceiverMode )`	
Parameter	
`TransceiverId [in]`	Abstracted CanIf TransceiverId which is assigned to a CAN transceiver, which controller mode has been transitioned.
`TransceiverMode [in]`	Transceiver mode to which the CAN transceiver transitioned.
Return code	
-	-
Functional Description	
Called by the CAN Transceiver Driver to notify CAN Interface about a successful transceiver mode transition at the specified CAN transceiver.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is reentrant. > Configuration Variant(s): `CANIF_TRCV_HANDLING == STD_ON`	
Call context	
> ANY	

Table 4-43 CanIf_TrcvModeIndication

4.3.9 CanIf_ControllerModeIndication

Prototype	
`void CanIf_ControllerModeIndication ( uint8 ControllerId, CanIf_ControllerModeType ControllerMode )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf ControllerId which is assigned to a CAN controller, which controller mode has been transitioned.
`ControllerMode [in]`	Controller mode to which the CAN controller transitioned.
Return code	
-	-
Functional Description	
Called by the CAN Driver of EQC AR 4.0.3, AR 4.2.1 or MSRC to notify CAN Interface about a successful controller mode transition at the specified CAN controller.	
Particularities and Limitations	
> Service ID: see table 'Service IDs'	
> This function is synchronous.	
> This function is reentrant only for different CAN controllers (`ControllerId`).	
> Contained in CanIf_Cfg.h, if a CAN Driver of EQC AR 4.0.3, AR 4.2.1 or MSRC is configured.	
> Configuration Variant(s): -	
Call context	
> ANY	

Table 4-44 CanIf_ControllerModeIndication

4.3.10 CanIf_ControllerModeIndication (CAN Driver of EQC AR 4.3.1)

Prototype	
`void CanIf_ControllerModeIndication ( uint8 ControllerId, Can_ControllerStateType ControllerMode )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf ControllerId which is assigned to a CAN controller, which controller mode has been transitioned.
`ControllerMode [in]`	Controller mode to which the CAN controller transitioned.
Return code	
-	-
Functional Description	
Called by the CAN Driver of EQC AR 4.3.1 to notify CAN Interface about a successful controller mode transition at the specified CAN controller.	
Particularities and Limitations	
> Service ID: see table 'Service IDs'	
> This function is synchronous.	
> This function is reentrant only for different CAN controllers (`ControllerId`).	
> Contained in CanIf_Cfg.h, if a CAN Driver of EQC AR 4.3.1 is configured.	
> Configuration Variant(s): -	
Call context	
> ANY	

Table 4-45 CanIf_ControllerModeIndication (CAN Driver of EQC AR 4.3.1)

4.3.11 CanIf_ConfirmPnAvailability

Prototype	
`void CanIf_ConfirmPnAvailability ( uint8 TransceiverId )`	
Parameter	
`TransceiverId [in]`	Abstracted CanIf TransceiverId which is assigned to a CAN transceiver, which has confirmed the PN availability.
Return code	
-	-
Functional Description	
Called by the CAN Transceiver Driver to notify CAN Interface that the specified CAN transceiver is running in PN communication mode.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is reentrant. > Configuration Variant(s): `CANIF_TRCV_HANDLING == STD_ON` and `CANIF_PN_TRCV_HANDLING == STD_ON`	
Call context	
> ANY	

Table 4-46 CanIf_ConfirmPnAvailability

4.3.12 CanIf_ClearTrcvWufFlagIndication

Prototype	
`void` **`CanIf_ClearTrcvWufFlagIndication`** `( uint8 TransceiverId )`	
Parameter	
`TransceiverId [in]`	Abstracted CanIf TransceiverId which is assigned to a CAN transceiver, which WUF flag was cleared.
Return code	
-	-
Functional Description	
Called by the CAN Transceiver Driver to notify CAN Interface that the specified CAN transceiver has cleared the WUF flag.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is reentrant. > Configuration Variant(s): `CANIF_TRCV_HANDLING == STD_ON` and `CANIF_PN_TRCV_HANDLING == STD_ON`	
Call context	
> ANY	

Table 4-47 CanIf_ClearTrcvWufFlagIndication

4.3.13 CanIf_CheckTrcvWakeFlagIndication

Prototype	
`void` **`CanIf_CheckTrcvWakeFlagIndication`** `( uint8 TransceiverId )`	
Parameter	
`TransceiverId [in]`	Abstracted CanIf TransceiverId which is assigned to a CAN transceiver, which has detected a wake-up.
Return code	
-	-
Functional Description	
Called by the CAN Transceiver Driver to notify CAN Interface that the specified CAN transceiver has finished the check of its wakeup events.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is reentrant. > Configuration Variant(s): `CANIF_TRCV_HANDLING == STD_ON` and `CANIF_PN_TRCV_HANDLING == STD_ON`	
Call context	
> ANY	

Table 4-48 CanIf_CheckTrcvWakeFlagIndication

4.3.14 CanIf_RamCheckCorruptMailbox

Prototype	
`void CanIf_RamCheckCorruptMailbox ( uint8 ControllerId, CanIf_HwHandleType HwHandle )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf ControllerId which is assigned to a CAN controller, with the corrupt mailbox.
`HwHandle [in]`	Handle of the corrupt mailbox.
Return code	
-	-
Functional Description	
Called by the CAN Driver to notify CAN Interface that the specified mailbox is corrupt. The specified corrupt mailbox will be notified to the application.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is non-reentrant. > Configuration Variant(s): `CANIF_EXTENDED_RAM_CHECK_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-49 CanIf_RamCheckCorruptMailbox

4.3.15 CanIf_RamCheckCorruptController

Prototype	
`void CanIf_RamCheckCorruptController ( uint8 ControllerId )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf ControllerId which is assigned to a CAN controller, which is corrupt.
Return code	
–	–
Functional Description	
Called by the CAN Driver to notify CAN Interface that the specified CAN controller is corrupt. The specified corrupt CAN controller will be notified to the application.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is non-reentrant. > Configuration Variant(s): `CANIF_EXTENDED_RAM_CHECK_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-50 CanIf_RamCheckCorruptController

4.3.16 CanIf_ControllerErrorStatePassive

Prototype	
`void CanIf_ControllerErrorStatePassive ( uint8 ControllerId, uint16 RxErrorCounter, uint16 TxErrorCounter )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf ControllerId which is assigned to a CAN controller, at which the transistion to error state passive occurred.
`RxErrorCounter [in]`	Value of the rx error counter from the specified CAN controller.
`TxErrorCounter [in]`	Value of the tx error counter from the specified CAN controller.
Return code	
–	–
Functional Description	
Called by the CAN Driver to notify the CAN Interface about an occurred transistion to error state passive at the specified CAN controller.	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is reentrant. > Configuration Variant(s): `CANIF_ENABLE_SECURITY_EVENT_REPORTING == STD_ON`	
Call context	
> ANY	

Table 4-51 CanIf_ControllerErrorStatePassive

4.3.17 CanIf_ErrorNotification

Prototype	
`void` **`CanIf_ErrorNotification`**`( uint8 ControllerId, Can_ErrorType CanError )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf ControllerId which is assigned to a CAN controller, at which the bus error occurred.
`CanError [in]`	Occured bus error.
Return code	
-	-
Functional Description	
Called by the CAN Driver to notify the CAN Interface about an occurred bus error at specified CAN controller	
Particularities and Limitations	
> Service ID: see table 'Service IDs’ > This function is synchronous. > This function is reentrant. > Configuration Variant(s): `CANIF_ENABLE_SECURITY_EVENT_REPORTING == STD_ON`	
Call context	
> ANY	

Table 4-52 CanIf_ErrorNotification

4.4 Configurable Interfaces

4.4.1 Notifications

At its configurable interfaces the CAN Interface defines notifications that can be mapped to callback functions provided by other modules. The mapping is not statically defined by the CAN Interface but can be performed at configuration time. The function prototypes that can be used for the configuration have to match the appropriate function prototype signatures, which are described in the following sub-chapters.

4.4.1.1 <User_RxIndication> (Simple Layout)

Prototype	
`void` **`<User_RxIndication>`** `( PduIdType CanRxPduId, const uint8* CanSduPtr )`	
Parameter	
`CanRxPduId [in]`	Upper layer handle from the received Rx-PDU.
`CanSduPtr [in]`	Pointer to the data of the received Rx-PDU.
Return code	
-	-
Functional Description	
Called by the CAN Interface to notify the configured upper layer about the reception of the specified Rx-PDU.	
Particularities and Limitations	
> Configuration Variant(s): `CANIF_RX_INDICATION_TYPE_I_IS_USED == STD_ON`	
Call context	
> ANY	

Table 4-53 <User_RxIndication> (Simple Layout)

4.4.1.2 <User_RxIndication> (Advanced Layout)

Prototype	
`void <User_RxIndication> ( PduIdType CanRxPduId, const PduInfoType* PduInfoPtr )`	
Parameter	
`CanRxPduId [in]`	Upper layer handle from the received Rx-PDU.
`PduInfoPtr [in]`	Pointer to a struct containing the properties of the received Rx-PDU.
Return code	
-	-
Functional Description	
Called by the CAN Interface to notify the configured upper layer about the reception of the specified Rx-PDU.	
Particularities and Limitations	
> Configuration Variant(s): -	
Call context	
> ANY	

Table 4-54 <User_RxIndication> (Advanced Layout)

4.4.1.3 <User_RxIndication> (NmOsek Layout)

Prototype	
`void <User_RxIndication> ( PduIdType CanRxPduId, const uint8* CanSduPtr, Can_IdType CanId )`	
Parameter	
`CanRxPduId [in]`	Upper layer handle from the received Rx-PDU.
`CanSduPtr [in]`	Pointer to the data of the received Rx-PDU.
`CanId [in]`	Can Id from the received Rx-PDU.
Return code	
-	-
Functional Description	
Called by the CAN Interface to notify the configured upper layer about the reception of the specified Rx-PDU.	
Particularities and Limitations	
> Configuration Variant(s): `CANIF_SUPPORT_NMOSEK_INDICATION == STD_ON`	
Call context	
> ANY	

Table 4-55 <User_RxIndication> (NmOsek Layout)

4.4.1.4 <User_RxIndication> (Cdd Layout)

Prototype	
`void <User_RxIndication> ( PduIdType CanRxPduId, const PduInfoType* PduInfoPtr, Can_IdType CanId )`	
Parameter	
`CanRxPduId [in]`	Upper layer handle from the received Rx-PDU.
`PduInfoPtr [in]`	Pointer to a struct containing the properties of the received Rx-PDU.
`CanId [in]`	Can Id from the received Rx-PDU.
Return code	
-	-
Functional Description	
Called by the CAN Interface to notify the configured upper layer about the reception of the specified Rx-PDU.	
Particularities and Limitations	
> Configuration Variant(s): `CANIF_RX_INDICATION_TYPE_IV_IS_USED == STD_ON`	
Call context	
> ANY	

Table 4-56 <User_RxIndication> (Cdd Layout)

4.4.1.5 <User_TxConfirmation>

Prototype	
`void <User_TxConfirmation> ( PduIdType TxPduId )`	
Parameter	
`TxPduId [in]`	Upper layer handle from the transmitted Tx-PDU.
Return code	
-	-
Functional Description	
Called by the CAN Interface to notify the configured upper layer about the transmission of the specified Tx-PDU.	
Particularities and Limitations	
> Configuration Variant(s): -	
Call context	
> ANY	

Table 4-57 <User_TxConfirmation>

4.4.1.6 <User_ControllerBusOff>

Prototype	
`void <User_ControllerBusOff> ( uint8 ControllerId )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf ControllerId which is assigned to a CAN controller at which a BusOff occurred.
Return code	
-	-
Functional Description	
Called by the CAN Interface to notify the configured upper layer about an occurred BusOff-event for the specified CAN controller.	
Particularities and Limitations	
> Configuration Variant(s): -	
Call context	
> ANY	

Table 4-58 <User_ControllerBusOff>

4.4.1.7 <User_ControllerModeIndication>

Prototype	
`void <User_ControllerModeIndication> ( uint8 ControllerId, CanIf_ControllerModeType ControllerMode )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf Controlle which is assigned to a CAN controller, at which the controller mode transition occurred.
`ControllerMode [in]`	Controller mode to which the CAN controller transitioned.
Return code	
-	-
Functional Description	
Called by the CAN Interface to notify the configured upper layer about a successful controller mode transition at the specified CAN controller.	
Particularities and Limitations	
> Configuration Variant(s): -	
Call context	
> ANY	

Table 4-59 <User_ControllerModeIndication>

4.4.1.8 <User_TrcvModeIndication>

Prototype	
`void <User_TrcvModeIndication> ( uint8 TransceiverId, CanTrcv_TrcvModeType TransceiverMode )`	
Parameter	
`TransceiverId [in]`	Abstracted TransceiverId which is assigned to a CAN transceiver, at which a transceiver mode transition occurred.
`TransceiverMode [in]`	Transceiver mode to which the CAN transceiver transitioned.
Return code	
-	-
Functional Description	
Called by the CAN Interface to notify the configured upper layer about a successful transceiver mode transition at the specified CAN transceiver.	
Particularities and Limitations	
> Configuration Variant(s): `CANIF_TRCV_HANDLING == STD_ON`	
Call context	
> ANY	

Table 4-60 <User_TrcvModeIndication>

4.4.1.9 <User_ConfirmPnAvailability >

Prototype	
`void <User_ConfirmPnAvailability> ( uint8 TransceiverId )`	
Parameter	
`TransceiverId [in]`	Abstracted CanIf TransceiverId which is assigned to a CAN transceiver, which has confirmed the PN availability.
Return code	
-	-
Functional Description	
Called by the CAN Interface to notify the configured upper layer that the specified CAN transceiver is running in PN communication mode.	
Particularities and Limitations	
> Configuration Variant(s): `CANIF_TRCV_HANDLING == STD_ON` and `CANIF_PN_TRCV_HANDLING == STD_ON`	
Call context	
> ANY	

Table 4-61 <User_ConfirmPnAvailability >

4.4.1.10 <User_ClearTrcvWufFlagIndication>

Prototype	
`void <User_ClearTrcvWufFlagIndication> ( uint8 TransceiverId )`	
Parameter	
`TransceiverId [in]`	Abstracted CanIf TransceiverId which is assigned to a CAN transceiver, which WUF flag was cleared.
Return code	
-	-
Functional Description	
Called by the CAN Interface to notify the configured upper layer that the specified CAN transceiver has cleared the WUF flag.	
Particularities and Limitations	
> Configuration Variant(s): `CANIF_TRCV_HANDLING == STD_ON` and `CANIF_PN_TRCV_HANDLING == STD_ON`	
Call context	
> ANY	

Table 4-62 <User_ClearTrcvWufFlagIndication>

4.4.1.11 <User_CheckTrcvWakeFlagIndication>

Prototype	
`void <User_CheckTrcvWakeFlagIndication> ( uint8 TransceiverId )`	
Parameter	
`TransceiverId [in]`	Abstracted CanIf TransceiverId which is assigned to a CAN transceiver, which has detected a wakeup.
Return code	
-	-
Functional Description	
Called by the CAN Interface to notify the configured upper layer that the specified CAN transceiver has finished the check of its wakeup events.	
Particularities and Limitations	
> Configuration Variant(s): `CANIF_TRCV_HANDLING == STD_ON` and `CANIF_PN_TRCV_HANDLING == STD_ON`	
Call context	
> ANY	

Table 4-63 <User_CheckTrcvWakeFlagIndication>

4.4.1.12 <User_ValidateWakeupEvent>

Prototype	
void <User_ValidateWakeupEvent> (EcuM_WakeupSourceType sources)	
Parameter	
sources [in]	Validated CAN wakeup events. Every CAN controller or CAN transceiver can be a separate wakeup source.
Return code	
-	-
Functional Description	
Called by the CAN Interface to notify the configured upper layer about the specified validated CAN wakeup events.	
Particularities and Limitations	
> Configuration Variant(s): CANIF_WAKEUP_VALIDATION == STD_ON	
Call context	
> ANY	

Table 4-64 <User_ValidateWakeupEvent>

4.4.1.13 <User_RamCheckCorruptMailbox>

Prototype	
void **<User_RamCheckCorruptMailbox>** (uint8 ControllerId, CanIf_HwHandleType HwHandle)	
Parameter	
ControllerId [in]	Abstracted CanIf Controller which is assigned to a CAN controller with the corrupt mailbox.
HwHandle [in]	The corrupt mailbox.
Return code	
-	-
Functional Description	
Called by the CAN Interface to notify the configured upper layer that the specified mailbox is corrupt.	
Particularities and Limitations	
> Configuration Variant(s): CANIF_EXTENDED_RAM_CHECK_SUPPORT == STD_ON	
Call context	
> ANY	

Table 4-65 <User_RamCheckCorruptMailbox>

4.4.1.14 <User_RamCheckCorruptController>

Prototype	
`void <User_RamCheckCorruptController> ( uint8 ControllerId )`	
Parameter	
`ControllerId [in]`	Abstracted CanIf Controller which is assigned to a CAN controller, which is corrupt.
Return code	
–	–
Functional Description	
Called by the CAN Interface to notify the configured upper layer that the specified CAN controller is corrupt.	
Particularities and Limitations	
> Configuration Variant(s): `CANIF_EXTENDED_RAM_CHECK_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-66 <User_RamCheckCorruptController>

4.4.1.15 <User_DataChecksumRxErrFct>

Prototype	
`void <User_DataChecksumRxErrFct> ( PduIdType CanIfRxPduId )`	
Parameter	
`CanIfRxPduId [in]`	Handle of the Rx-PDU which has a wrong data chechsum.
Return code	
–	–
Functional Description	
Called by the CAN Interface to notify the configured upper layer that the specified Rx-PDU was received with a wrong data checksum.	
Particularities and Limitations	
> Configuration Variant(s): `CANIF_DATA_CHECKSUM_RX_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-67 <User_DataChecksumRxErrFct>

4.4.2 Callout Functions

At its configurable interfaces the CAN Interface defines callout functions. The declarations of the callout functions are provided by the BSW module, i.e. the CAN Interface It is the integrator's task to provide the corresponding function definitions. The definitions of the callouts can be adjusted to the system's needs. The CAN Interface callout function declarations are described in the following tables

4.4.2.1 CanIf_RxIndicationSubDataChecksumRxVerify

Prototype	
`Std_ReturnType` **`CanIf_RxIndicationSubDataChecksumRxVerify`** `( PduIdType CanIfRxPduId, Can_IdType CanId, uint8 CanDlc, const uint8 *CanSduPtr )`	
Parameter	
`CanIfRxPduId [in]`	Handle of the Rx-PDU, which checksum shall be verifed.
`CanId [in]`	CAN identifier of received Rx-PDU
`CanDlc [in]`	Data length of received Rx-PDU
`CanSduPtr [in]`	Pointer to data of received Rx-PDU
Return code	
`E_OK`	Verification of checksum passed. In this case the Rx-PDU is forwarded to upper layer.
`E_NOT_OK`	Verification of checksum failed. In this case the Rx-PDU is discarded and NOT forwarded to upper layer.
Functional Description	
Is called by the CAN Interface to verify the data checksum from a received Rx-PDU that contains a data checksum.	
Particularities and Limitations	
> This function is synchronous. > This function is reentrant. > This API is called in context in which the consistency of all passed parameters is ensured. Hence no further protection is required within. > Configuration Variant(s): `CANIF_DATA_CHECKSUM_RX_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-68 CanIf_RxIndicationSubDataChecksumRxVerify

4.4.2.2 CanIf_TransmitSubDataChecksumTxAppend

Prototype	
`void TransmitSubDataChecksumTxAppend ( const Can_PduType *txPduInfoPtr, uint8 *txPduDataWithChecksumPtr )`	
Parameter	
`txPduInfoPtr [in]`	Pointer to Tx-PDU-parameters: CAN identifier, data length, data.
`txPduDataWithChecksumPtr [out]`	Pointer to data buffer where the data of Tx-PDU incl. the checksum will be stored in. The data checksum PDU is transmitted with data stored in this buffer. Note: Parameter "`txPduDataWithChecksumPtr`" may only be written with index >= 0 and < `CANIF_CFG_MAXTXDLC_PLUS_DATACHECKSUM` (see file `CanIf_Cfg.h`). The length of data can not be changed hence the checksum must only be added within valid data-length of the Tx-PDU which is given by range: 0 - (`txPduInfoPtr->length` - 1).
Return code	
–	–
Functional Description	
Is called by the CAN Interface before transmission of a data checksum Tx-PDU in order to add a checksum to its data.	
Particularities and Limitations	
> This function is synchronous. > This function is reentrant. > This API is called in context in which the consistency of all passed parameters is ensured. Hence no further protection is required within. > Configuration Variant(s): `CANIF_DATA_CHECKSUM_TX_SUPPORT == STD_ON`	
Call context	
> ANY	

Table 4-69 CanIf_TransmitSubDataChecksumTxAppend

4.4.2.3 <Appl_CanIfSetDynamicRxId>

Prototype	
`void` **`<Appl_CanIfSetDynamicRxId>`** `(PduIdType RxPduId, Can_IdType* CanId)`	
Parameter	
`RxPduId [in]`	PDU ID of the dynamic Rx PDU. ID is defined as “AUTOSAR Rx PDU handles” in CanIf_Cfg.h
`CanId [out]`	Pointer to a memory location where the CAN ID to be set must be stored.
Return code	
-	-
Functional Description	
Is called by the CAN Interface during `CanIf_Init` for a dynamic Rx PDU to set the CAN ID.	
Particularities and Limitations	
> This function is synchronous. > This function is reentrant. > This API is called in context in which the consistency of all passed parameters is ensured. Hence no further protection is required within. > Configuration Variant(s): `CANIF_SETDYNAMICRXID_API== STD_ON`	
Call context	
> ANY	

Table 4-70 <Appl_CanIfSetDynamicRxId>

5 Configuration

In the CAN Interface the attributes can be configured with the following tool:

- DaVinci Configurator Classic

5.1 Configuration Variants

The CAN Interface supports the configuration variants

- `VARIANT-PRE-COMPILE`
- `VARIANT-POST-BUILD-LOADABLE`
- `VARIANT-POST-BUILD-SEELCTABLE`

The configuration classes of the CAN Interface parameters depend on the supported configuration variants. For their definitions please see the `CanIf_bswmd.arxml` file.

6 Glossary and Abbreviations

6.1 Glossary

Term	Description
DaVinci Configurator Classic	Configuration and generation tool for MICROSAR Classic software components

Table 6-1 Glossary

6.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
Appl	Application
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
BSWMD	Basis Software Module Description
CAN	Controller Area Network
CanIf	CAN Interface module
CanNm	CAN Network Manager module
CanSM	CAN State Manager module
CanTp	CAN Transport Layer module
CanTSyn	Global Time Synchronization over CAN module
CanTrcv	CAN Transceiver Driver module
CCP	CAN Calibration Protocol module
CDD	Complex Device Driver module
DET	Default Error Tracer module
DLC	Data Length Code
ECU	Electronic Control Unit
EcuM	ECU State Manager module
EQC	Equivalence Class
FD	Flexible Data-rate
FIFO	First In First Out
HRH	Hardware Receive Handle
HTH	Hardware Transmit Handle
HW	Hardware
IdsM	Intrusion Detection System Manager module
J1939Nm	J1939 Network Management module
J1939Tp	J1939 Transport Layer module
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
Msn	Module short name of a BSW component/module

Abbreviation	Description
MSRC	MICROSAR Classic
PDU	Protocol Data Unit
PduR	PDU Router module
SchM	Schedule Manager module
SDU	Service Data Unit
SRS	Software Requirement Specification
SWS	Software Specification
VCID	Virtual CAN network ID
VStdLib	Vector Standard Library
XCP	Universal Calibration Protocol module

Table 6-2 Abbreviations

7 Contact

Visit our website for more information on

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

www.vector.com