Document Title	Specification of CAN Interface
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	012
Document Classification	Standard

Document Version	5.0.0
Document Status	Final
Part of Release	4.0
Revision	3

Document Change History

Date	Version	Changed by	Change Description
01.12.2011	5.0.0	AUTOSAR Administration	• Partial Networking Support • Improved Transmit Buffering • Improved Error Detection
22.10.2010	4.1.0	AUTOSAR Administration	• Updated chapters "Version Checking" and "Published Information" • Multiple CAN IDs could optionally be assigned to one I-PDU • Wake-up validation optionally only via NM PDUs • Asynch. mode indication call-backs instead of synch. mode changes • No automatic PDU channel mode change when CC mode changes • TxConfirmation state entered for BusOff Recovery • WakeupSourceRefIn and WakeupSourceRefOut • PduInfoPtr instead of SduDataPtr • Introduction of Can_GeneralTypes.h and Can_HwHandleType • Transceiver types of chapter 8. shifted to transceiver SWS

Document Change History

Date	Version	Changed by	Change Description
02.12.2009	4.0.0	AUTOSAR Administration	• HOH definition • abstracted ControllerId and TransceiverId • No changing of baudrate via CanIf and CanIf_ControllerInit • Dispatcher adapted because of CDD • TxBuffering: only one buffer per L-PDU • Wake up mechanism adapted to environment behavior (network -> controller/transceiver; wakeupSource) • Mode changes made asynchronous • no complete state machine in CanIf, just buffered states per controller • Legal disclaimer revised
23.06.2008	3.0.2	AUTOSAR Administration	Legal disclaimer revised
29.01.2008	3.0.1	AUTOSAR Administration	• Replaced chapter 10 content with generated tables from AUTOSAR MetaModel.
12.12.2007	3.0.0	AUTOSAR Administration	• Interface abstraction: network related interface changed into a controller related one • Wakeup mechanism completely reworked, APIs added & changed for Wakeup • Initialization changed (flat initialization) • Scheduled main functions skipped due to changed BSW Scheduler responsibility • Document meta information extended • Small layout adaptations made

Document Change History			
Date	**Version**	**Changed by**	**Change Description**
31.10.2007	2.1.0	AUTOSAR Administration	• Header file structure changed • Support of mixed mode operation (Standard CAN & Extended CAN in parallel on one network) added • Support of CAN Transceiver API <User>_DlcErrorNotification deleted • Pre-compile/Link-Time/Post-Built definition for configuration parameters partly changed • Re-entrant interface call allowed for certain APIs • Support of AUTOSAR BSW Scheduler added • Support of memory mapping added • Configuration container structure reworked • Various of clarification extensions and corrections
26.06.2006	2.0.0	AUTOSAR Administration	Second Release
31.06.2005	1.0.0	AUTOSAR Administration	Initial Release

Disclaimer

This specification and the material contained in it, as released by AUTOSAR is for the purpose of information only. AUTOSAR and the companies that have contributed to it shall not be liable for any use of the specification.

The material contained in this specification is protected by copyright and other types of Intellectual Property Rights. The commercial exploitation of the material contained in this specification requires a license to such Intellectual Property Rights.

This specification may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only.
For any other purpose, no part of the specification may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The AUTOSAR specifications have been developed for automotive applications only. They have neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Advice for users

AUTOSAR Specification Documents may contain exemplary items (exemplary reference models, "use cases", and/or references to exemplary technical solutions, devices, processes or software).

Any such exemplary items are contained in the Specification Documents for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such Specification Documents, nor any later documentation of AUTOSAR conformance of products actually implementing such exemplary items, imply that intellectual property rights covering such exemplary items are licensed under the same rules as applicable to the AUTOSAR Standard.

Table of Contents

Known Limitations

- The parameter wakeupSource used in the wake up mechanism (CanIf_CheckWakeup, <User_ValidateWakeupEvent>, <User_SetWakeupEvent>, Can_CheckWakeup, CanTrcv_CheckWakeup) is not fully specified.

1 Introduction and functional overview

This specification describes the functionality, API and the configuration for the AUTOSAR Basic Software module CAN Interface.

The CAN Interface module is located between the low level CAN device drivers (CAN Driver and Transceiver Driver) and the upper communication service layers (i.e. CAN State Manager, CAN Network Management, CAN Transport Protocol, PDU Router). It represents the interface to the services of the CAN Driver for the upper communication layers.

The CAN Interface module provides a unique interface to manage different CAN hardware device types like CAN controllers and CAN transceivers used by the defined ECU hardware layout. Thus multiple underlying internal and external CAN controllers/CAN transceivers can be controlled by the CAN State Manager module based on a physical CAN channel related view.

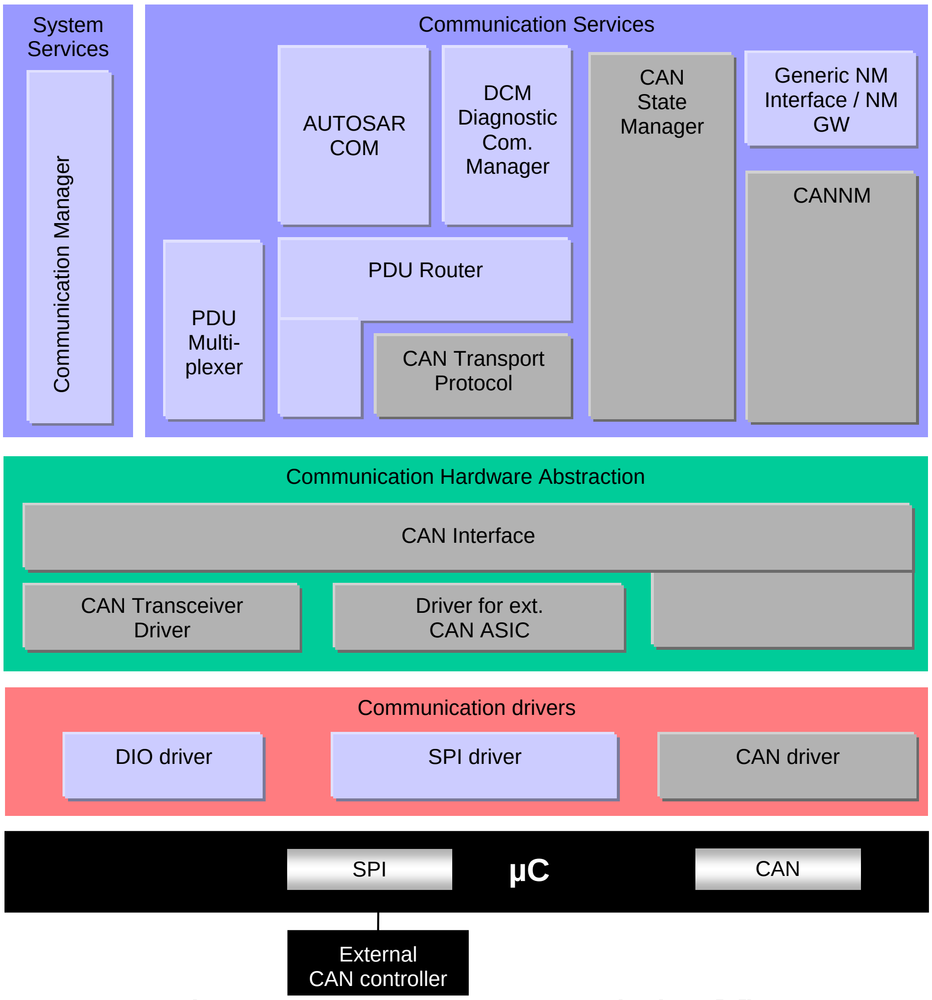


Figure 1 AUTOSAR CAN Layer Model (see [2])

The CAN Interface module consists of all CAN hardware independent tasks, which belongs to the CAN communication device drivers of the corresponding ECU. Those functionality is implemented once in the CAN Interface module, so that underlying CAN device drivers only focus on access and control of the corresponding specific CAN hardware device.

The CAN Interface module fulfils main control flow and data flow requirements of the PDU Router and upper layer communication modules of the AUTOSAR COM stack: transmit request processing, transmit confirmation / receive indication / error notification and start / stop of a CAN controller and thus waking up / participating on a network. Its data processing and notification API to is based on CAN L-PDUs, whereas APIs for control and mode handling provides a CAN controller related view.

In case of transmit requests the CAN Interface module completes the L-PDU transmission with corresponding parameters and relays the CAN L-PDU via the appropriate CAN Driver to the CAN controller. At reception the CAN Interface module distributes the received L-PDUs to the upper layer. The assignment between receive L-PDU and upper layer is statically configured. At transmit confirmation the CAN Interface is responsible for the notification of upper layers about successful transmission.

The CAN Interface module provides CAN communication abstracted access to the CAN Driver and CAN Transceiver Driver services for control and supervision of the CAN network. The CAN Interface forwards downwards the status change requests from the CAN State Manager to the lower layer CAN device drivers, and upwards the CAN Driver / CAN Transceiver Driver events are forwarded by the CAN Interface module to e.g. the corresponding NM module.

2 Acronyms and Abbreviations

The glossary below includes acronyms and abbreviations relevant to the CAN Interface module that are not included in the AUTOSAR glossary.

Abbreviation / Acronym:	***Description:***
CAN L-PDU	CAN Protocol Data Unit. Consists of an identifier, DLC and data (SDU).
CAN L-SDU	CAN Service Data Unit. Data that are transported inside the CAN L-PDU.
CanDrv	CAN Driver module
CanIf	CAN Interface module
CanNm	CAN Network Management module
CanSm	CAN State Manager module
CanTp	CAN Transport Layer module
CanTrcv	CAN Transceiver Driver module
CCMSM	CAN Interface Controller Mode State Machine (for one controller)
CDD	Complex Device Driver
ComM	Communication Manager module
DCM	Diagnostic Communication Manager module
Dem	Diagnostic Event Manager module
DET	Development Error Tracer module
DLC	Data Length
DLL	Data Link Layer
EcuM	ECU State Manager module
FIFO	First-In-First-Out
HOH	CAN hardware object handle
HRH	CAN hardware receive handle
HTH	CAN hardware transmit handle
ISR	Interrupt service routine
L-PDU	Protocol Data Unit for the data link layer (DLL)
L-SDU	Service Data Unit for the data link layer (DLL)
PDU	Protocol Data Unit
PduR	PDU Router module
PN	Partial Networking
SDU	Service Data Unit

Terms:	***Description:***
Buffer	Fixed sized memory area for a single data unit (e.g. CAN ID, DLC, SDU, etc.) is stored at a dedicated memory address in RAM.
CAN communication matrix	Describes the complete CAN network: ▪ Participating nodes ▪ Definition of all CAN PDUs (identifier, DLC) ▪ Source and Sinks for PDUs
CAN controller	A CAN controller is a CPU on-chip or external standalone hardware device. One CAN controller is connected to one physical channel.
CAN device driver	Generic term of CAN Driver and CAN Transceiver Driver.
CAN hardware unit	A CAN Hardware unit may consist of one or multiple CAN controllers of the same type and one, two or multiple CAN RAM areas. The CAN hardware unit is located on-chip or as external device. The CAN hardware unit is represented by one CAN Driver.
CanIf Controller mode state machine	This is not really a state machine, which may be influenced by transmission requests. This is an image of the current abstracted state of an appropriate CAN controller. The state transitions can only be realized by UL modules like the CanSm or by external events like e.g. if a BusOff occurred.

CanIf Receive L-PDU / CanIf Rx L-PDU	L-PDU handle of which the direction is set to “lower to upper layer”.
CanIf Receive L-PDU buffer / CanIfRxBuffer	Single element RAM buffer located in the CAN Interface module to store whole receive L-PDUs.
CanIf Transmit L-PDU / CanIf Tx L-PDU	L-PDU handle of which the direction is set to “upper to lower layer”.
CanIf Transmit L-PDU buffer / CanIfTxBuffer	Single CanIfTxBuffer element located in the CanIf to store one or multiple CanIf Tx L-PDUs. If the buffersize of a single CanIfTxBuffer element is set to 0, a CanIfTxBuffer element is only used to refer a HTH.
Hardware object/ HW object	A CAN hardware object is defined as a PDU buffer inside the CAN RAM of the CAN hardware unit / CAN controller.
Hardware receive handle (HRH)	The Hardware Receive Handle (HRH) is defined and provided by the CAN Driver. Each HRH typically represents just one hardware object. The HRH is used as a parameter by the CAN Interface Layer for i.e. software filtering.
Hardware transmit handle (HTH)	The Hardware Transmit Handle (HTH) is defined and provided by the CAN Driver. Each HTH typically represents just one or multiple CAN hardware objects that are configured as CAN hardware transmit buffer pool.
Inner priority inversion	Transmission of a high-priority L-PDU is prevented by the presence of a pending low-priority L-PDU in the same transmit hardware object.
Integration Code	Code that the Integrator needs to add to an AUTOSAR System, to adapt non-standardized functionalities. Examples are Callouts of the ECU State Manager and Callbacks of various other BSW modules. The I/O Hardware Abstraction is called Integration Code, too.
Lowest In – First Out / LOFO	This is a data storage procedure, whereas always the elements with the lowest values will be extracted.
L-PDU handle	The L-PDU handle is defined as integer type and placed inside the CAN Interface layer. Typically, each handle represents an L-PDU, which is a constant structure with information for Tx/Rx processing.
L-PDU channel group	Group of CAN L-PDUs, which belong to just one underlying network. Usually they are handled by one upper layer module.
Outer priority inversion	A time gap occurs between two consecutive transmit L-PDUs. In this case a lower priority L-PDU from another node can prevent sending the own higher priority L-PDU. Here the higher priority L-PDU cannot participate in arbitration during network access because the lower priority L-PDU already won the arbitration.
Physical channel	A physical channel represents an interface from a CAN controller to the CAN Network. Different physical channels of the CAN hardware unit may access different networks.
Tx request	Transmit request to the CAN Interface module from a upper layer module of the CanIf

3 Related documentation

3.1 Input documents

[1] List of Basic Software Modules
AUTOSAR_TR_BSWModuleList.pdf

[2] Layered Software Architecture
AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf

[3] General Requirements on Basic Software Modules
AUTOSAR_SRS_BSWGeneral.pdf

[4] Specification of Standard Types
AUTOSAR_SWS_StandardTypes.pdf

[5] Specification of Communication Stack Types
AUTOSAR_SWS_CommunicationStackTypes.pdf

[6] Specification of ECU Configuration
AUTOSAR_TPS_ECUConfiguration.pdf

[7] Requirements on CAN
AUTOSAR_SRS_CAN.pdf

[8] Specification of CAN Driver
AUTOSAR_SWS_CANDriver.pdf

[9] Specification of CAN Transceiver Driver
AUTOSAR_SWS_CANTransceiverDriver.pdf

[10] Specification of CAN Transport Layer
AUTOSAR_SWS_CANTransportLayer.pdf

[11] Specification of CAN State Manager
AUTOSAR_SWS_CAN_StateManager.pdf

[12] Specification of CAN Network Management
AUTOSAR_SWS_CAN_NM.pdf

[13] Specification of Generic Specification of Generic Network Management Interface
AUTOSAR_SWS_NetworkManagementInterface.pdf

[14] Specification of Communication
AUTOSAR_SWS_COM.pdf

[15] Specification of ECU State Manager
AUTOSAR_SWS_ECUStateManager.pdf

[16] Specification of BSW Scheduler
AUTOSAR_SWS_BSW_Scheduler.pdf

[17] Basic Software Module Description Template
AUTOSAR_TPS_BSWModuleDescriptionTemplate.pdf

3.2 Related standards and norms

[18] ISO11898 – Road vehicles - controller area network (CAN)

[19] ISO14229-1 Unified diagnostic services (UDS) - Part 1: Specification and Requirements (ISO DIS 26.05.2004)

[20] ISO15765-2 Diagnostics on controller area network (CAN) - Part 2: Network layer services

[21] ISO15765-3 Diagnostics on controller area network (CAN) - Part 3: Implementation of unified diagnostic services (UDS on CAN)

4 Constraints and assumptions

4.1 Limitations

The CAN Interface can be used for CAN communication only and is specifically designed to operate with one or multiple underlying CAN Drivers and CAN Transceiver Drivers. Several CAN Driver modules covering different CAN hardware units are represented by just one generic interface as specified in the CAN Driver specification. As well in the same manner several CAN Transceiver Driver modules covering different CAN transceiver devices are represented by just one generic interface as specified in the CAN Transceiver Driver specification. Other protocols than CAN (i.e. LIN or FlexRay) are not supported.

4.2 Applicability to car domains

The CAN Interface can be used for all domain applications when the CAN protocol is used.

5 Dependencies to other modules

This section describes the relations to other modules within the AUTOSAR basic software architecture. It contains brief descriptions of configuration information and services, which are required by the CAN Interface Layer from other modules.

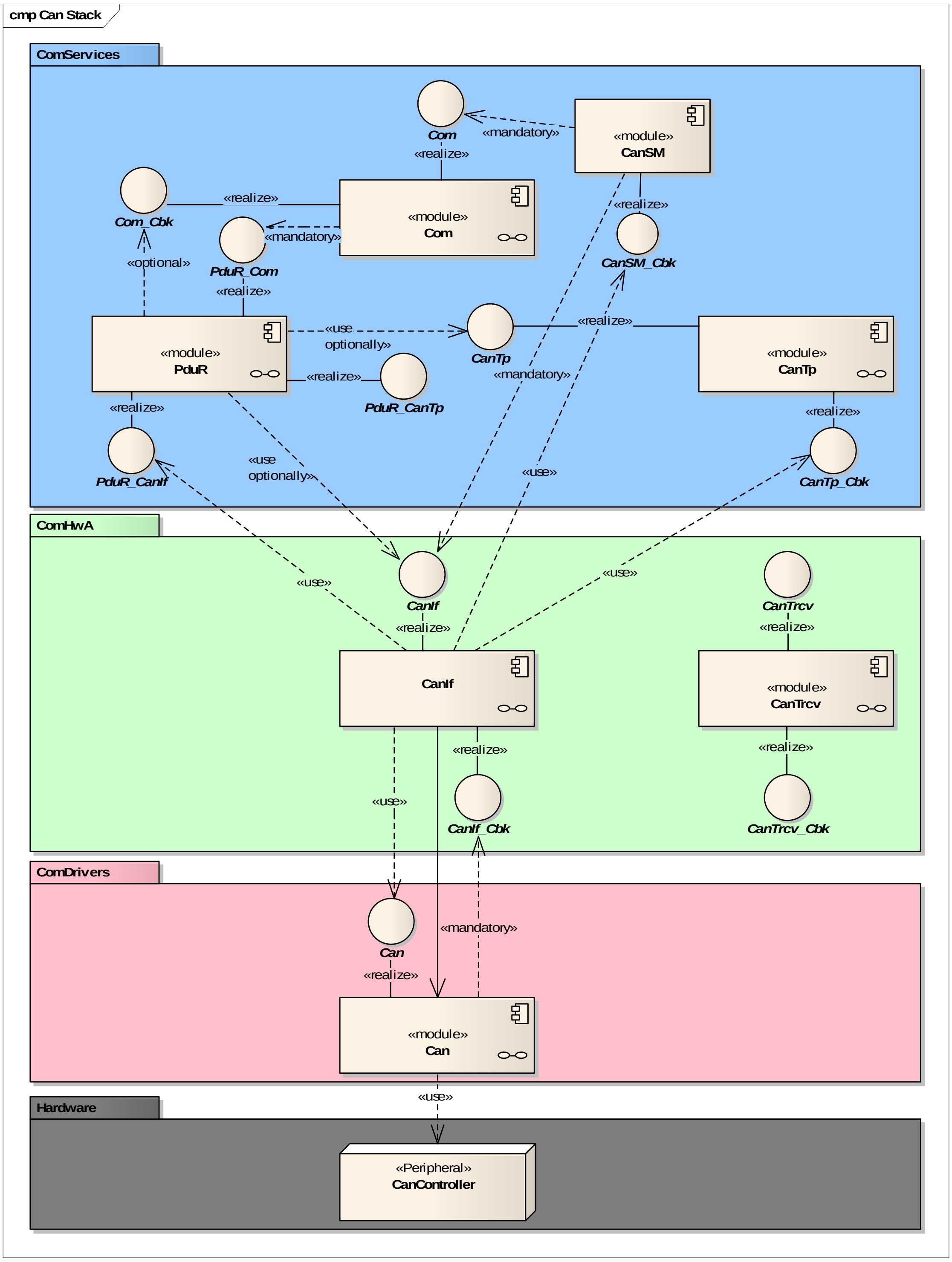


Figure 2 CANIF dependencies in AUTOSAR BSW

5.1 Upper Protocol Layers

Inside the AUTOSAR BSW architecture the upper layers of the CAN Interface module (Abbr.: CanIf) are represented by the PDU Router module (Abbr.: PduR), CAN Network Management module (Abbr.: CanNm), CAN Transport Layer module (Abbr.: CanTp), CAN State Manager module (Abbr.: CanSm), ECU State Manager module (abbr.: EcuM) and Complex Device Driver modules (Abbr.: CDD).

The AUTOSAR BSW architecture indicates that the application data buffers are located in the upper layer, to which they belong. Direct access to these buffers is prohibited. The buffer location is passed by the CanIf from or to the CAN Driver module (Abbr.: CanDrv) during transmission and reception. During execution of these transmission/reception indication services buffer location is passed. Data integrity is guaranteed by use of lock mechanisms each time the buffer has been accessed. See [7.18 Data integrity].

The API used by the CanIf consists of notification services as basic agents for the transfer of CAN related data (i.e. CAN DLC) to the target upper layer. The call parameters of these services points to the information buffered in the CanDrv or they refer directly to the CAN hardware.

5.2 Initialization: Ecu State Manager

The EcuM initializes the CanIf (refer to [15] Specification of ECU State Manager).

5.3 Mode Control: CAN State Manager

The CanSM module is responsible for mode control management of all supported CAN controllers and CAN transceivers.

5.4 Lower layers: CAN Driver

The main lower layer CAN device driver is represented by the CanDrv (see [8] Specification of CAN Driver). The CanIf has a close relation to the CanDrv as a result of its position in the AUTOSAR Basic Software Architecture.

The CanDrv provides a hardware abstracted access to the CAN controller only, but control of operation modes is done in CanSm only.

The CanDrv detects and processes events of the CAN controllers and notifies those to the CanIf.

The CanIf passes operation mode requests of the CanSm to the corresponding underlying CAN controllers.

The CanDrv provides a normalized L-SDU to ensure hardware independence of the CanIf. The pointer to this normalized L-SDU points either to a temporary buffer (for e.g. data normalizing) or to the CAN hardware dependent to the CanDrv. For the CanIf the kind of L-SDU buffer is invisible.

The CanIf provides notification services used by the CanDrv in all notifications scenarios, for example: transmit confirmation (8.4.1 CanIf_TxConfirmation, see [CANIF007), receive indication (8.4.2 CanIf_RxIndication, see [CANIF006), transmit cancellation notification (8.4.3 CanIf_CancelTxConfirmation, see [CANIF101), BusOff notification (8.4.4 CanIf_ControllerBusOff, see [CANIF218) and notification of a controller mode change (8.4.8, see CANIF669).

In case of using multiple CanDrv serving different interrupt vectors these callback services mentioned above must be re-entrant, refer to [7.25 Multiple CAN Driver support]. Reentrancy of callback functions is specified in chapter 8.4.

The callback services called by the CanDrv are declared and implemented inside the CanIf. The callback services called by the CanIf are declared and placed inside the appropriate upper communication service layer, for example PduR, CanNm, CanTp. The CanIf structure is specified in chapter 5.7 File structure.

The number of configured CAN controllers does not necessarily belong to the number of used CAN transceivers. In case multiple CAN controllers of a different types operate on the same CAN network, one CAN transceiver and CanTrcv is sufficient, whereas dependent to the type of the CAN controller devices one or two different CanDrv are needed (see 7.5 Physical channel view).

5.5 Lower layers: CAN Transceiver Driver

The second available lower layer CAN device driver is represented by the CanTrcv (see [9] Specification of CAN Transceiver Driver) (Abbr.: CanTrcv).

Each CanTrcv itself does operation mode control of the CAN transceiver device. The CanIf just maps all APIs of several underlying Cantrcv to a unique one, thus CanSm is able to trigger a transition of the corresponding CAN transceiver modes. No control or handling functionality belonging to CanTrcv is done inside the CanIf.

The CanIf maps the following services of all underlying CanTrcvs to one unique interface. These are further described in the CAN Transceiver Driver SWS (see [9]Specification of CAN Transceiver Driver):

- Unique CanTrcv mode request and read services to manage the operation modes of each underlying CAN transceiver device.
- Read service for CAN transceiver wake up reason support.
- Mode request service to enable/disable/clear wake up event state of each used CAN transceiver (`CanIf_SetTrcvMode()`, see CANIF287).

5.6 Configuration

The CanIf design is optimized to manage CAN protocol specific capabilities and handling of the used underlying CAN controller.

The CanIf is capable to change the CAN configuration without a re-build. Therefore the function CanIf_Init (see [CANIF001) retrieves the required CAN configuration information from configuration containers and parameters, which are specified (linked as references, or additional parameters) in chapter 10, see Figure 32 Overview about CAN Interface configuration containers
. This section gives a summary of the retrieved information, e.g.:

- Number of CAN controllers. The number of CAN controllers is necessary for dispatching of transmit and receive L-PDUs and for the control of the status of the available CAN Drivers (see CanIfCanControllerIdRef).
- Number of hardware object handles. To supervise transmit requests the CAN Interface needs to know the number of HTHs and the assignments between each HTH and the corresponding CAN controller (see CANIF_HTH_CAN_CONTROLLER_ID_REF, CANIF625_Conf; CANIF_HTH_ID_SYMREF, CANIF627_Conf).
- Range of received CAN IDs passing hardware acceptance filter for each hardware object. The CAN Interface uses fixed assignments between HRHs and L-PDUs to be received in the corresponding hardware object to conduct a search algorithm (see 7.21 Software receive filter, see CANIF_SOFTWARE_FILTER_HRH, CANIF_HRH_CAN_CONTROLLER_ID_REF, CANIF_HRH_ID_SYMREF, CANIF634_Conf)

The CanIf needs information about all used upper communication service layers and L-PDUs to be dispatched. The following information has to be set up at configuration time for integration of the CanIf inside the AUTOSAR COM stack:

- Transmitting upper layer module and transmit I-PDU for each transmit L-PDU.
  => Used for dispatching of transmit confirmation services (see CANIF_CANTXPDUID, CANIF247_Conf).
- Receiving upper layer module and receive I-PDU for each receive L-PDU.
  => Used for L-PDU dispatching during receive indication (see CANIF_CANRXPDUID, CANIF249_Conf).

The CanIf needs the description of the controller and the own ECU, which is connected to one or multiple CAN networks. The following information is therefore retrieved from the CAN communication matrix, part of the AUTOSAR system configuration (see containers: CanIfTxPduConfig, CANIF248_Conf; CanIfRxPduConfig, CANIF249_Conf):

- All L-PDUs received on each physical channel of this ECU.
  => Used for software filtering and receive L-PDU dispatch
- All L-PDUs that shall be transmitted by each physical channel on this ECU.
  => Used for the transmit request and transmit L-PDU dispatch
- Properties of these L-PDUs (ID, DLC).
  => Used for software filtering, receive indication services, DLC check
- Transmitter for each transmitted L-PDU (i.e. PduR, CanNm, CanTp).
  => Used for the transmit confirmation services
- Receiver for each receive L-PDU (i.e. PduR, CanNm, CanTp)
  => Used for the L-PDU dispatch

- Symbolic L-PDU name.
  => Used for the representation of Rx/Tx data buffer addresses

5.7 File structure

5.7.1 Code file structure

[CANIF374] ⌈The code file structure shall not be defined within this specification completely. Here it shall be pointed out that the code-file structure shall include the following files named:

- `CanIf_Lcfg.c` – for link time configurable parameters.
- `CanIf_PBcfg.c` – for post build time configurable parameters.

These files shall contain all link time and post-build time configurable parameters.⌋ (BSW00380)

[CANIF375] ⌈The code-file structure shall include `CanIf_<X>.c` – for implementation of the provided functionality. The extension <X> is optional for usage of multiple C-files.⌋()

[CANIF376] ⌈The code-file structure shall include `CanIf_Cfg.c` – for pre-compile time configurable parameters.⌋(BSW00380, BSW00419)

[CANIF377] ⌈The CanIf shall access the location of the API of all used underlying CanDrvs for pre-compile time configuration either by using of external declaration in includes of all CanDrvs public header files `can_<x>.h` or by the code file `CanIf_Cfg.c`. ⌋()

[CANIF378] ⌈The CanIf shall access the location of the API of all used underlying CanDrvs for link time configuration by a set of function pointers for each CanDrv.⌋()

The values for the function pointers for each CanDrv are given at link time.

Rationale for CANIF377 and CANIF378: The API of all used underlying CanDrv must be known at the latest at link-time.

The include file structure can be constructed as shown in figure 3.

5.7.2 Header file structure

[CANIF116] ⌈The CanIf shall offer a header file `CanIf.h`, which contains the declaration of the CanIf API.. ⌋()

[CANIF672] ⌈The header file `CanIf.h` only contains extern declarations of constants, global data and services that are specified in the CanIf SWS.⌋()

Constants, global data types and functions that are only used by the CanIf internally, are declared within `CanIf.c`.

[CANIF643] ⌈The generic type definitions of the CanIf which are described in chapter 8.2 shall be performed in the header file `CanIf_Types.h`. This file has to be included in the header file `CanIf.h`.⌋()

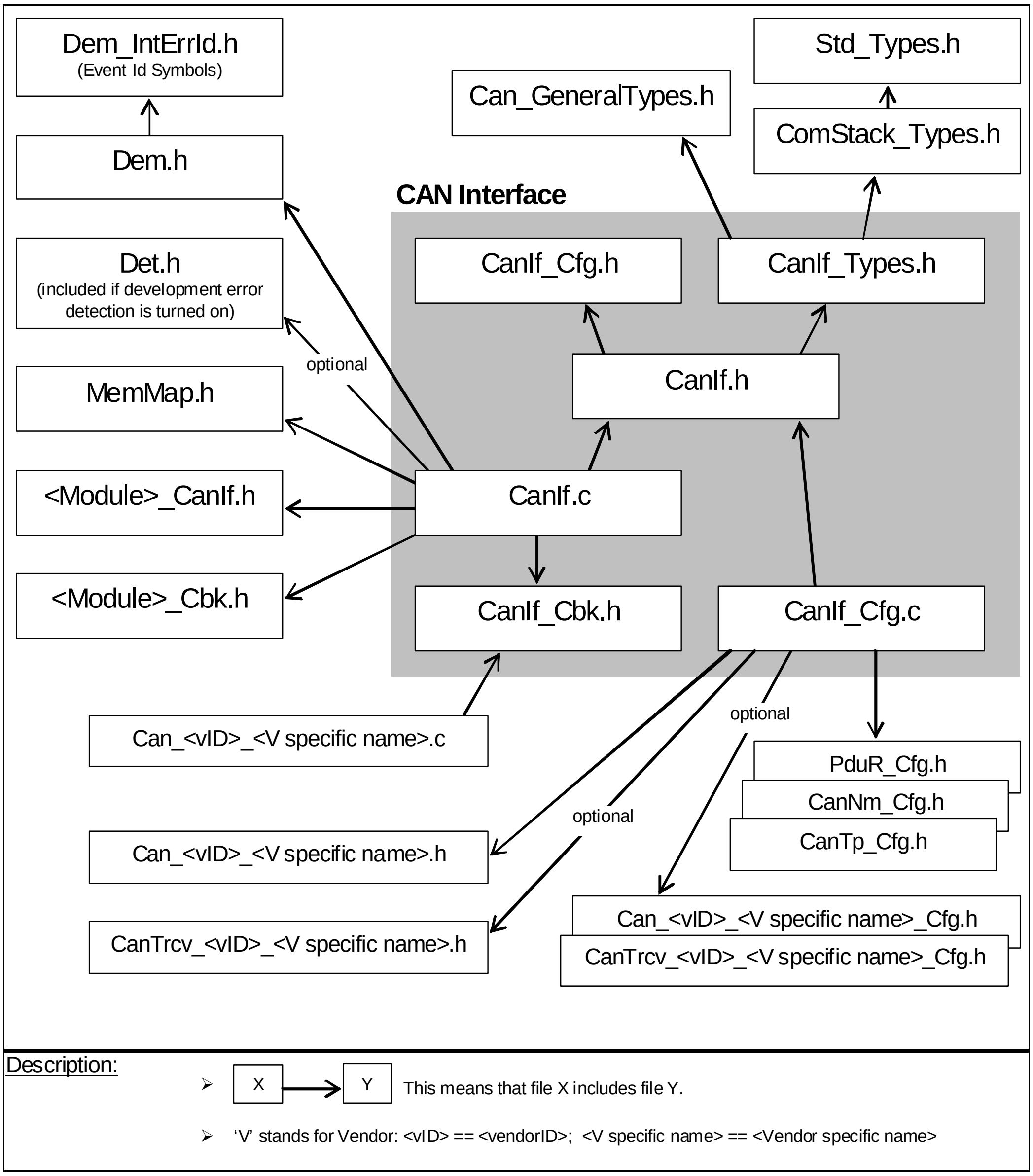

Dem_IntErrId.h
(Event Id Symbols)
Std_Types.h
Can_GeneralTypes.h
ComStack_Types.h
Dem.h
CAN Interface
Det.h
(included if development error detection is turned on)
CanIf_Cfg.h
CanIf_Types.h
optional
MemMap.h
CanIf.h
<Module>_CanIf.h
CanIf.c
<Module>_Cbk.h
CanIf_Cbk.h
CanIf_Cfg.c
optional
Can_<vID>_<V specific name>.c
PduR_Cfg.h
CanNm_Cfg.h
optional
CanTp_Cfg.h
Can_<vID>_<V specific name>.h
Can_<vID>_<V specific name>_Cfg.h
CanTrcv_<vID>_<V specific name>.h
CanTrcv_<vID>_<V specific name>_Cfg.h
Description:
X
Y
This means that file X includes file Y.
'V' stands for Vendor: <vID> == <vendorID>; <V specific name> == <Vendor specific name>

Figure 3 Code and include file structure

[CANIF121] ⌈The CanIf shall provide a header file `CanIf_Cbk.h`, which declares the callback functions called by the CanDrv.⌋()

[CANIF122] ⌈The CanIf shall include necessary configuration data by the header files:

- `CanIf.h` – for declaration of the provided interface functions
- `CanIf_Cfg.h` – for pre-compile time configurable parameters and
- `CanIf_Lcfg.h` – for link build time configurable parameters
- `CanIf_PBcfg.h` – for post build time configurable parameters⌋ (BSW00381, BSW00412)

[CANIF463] ⌈The CanIf include the following header files `<Module>.h:`

- `Can_<vendorID>_<Vendor specific name><driver abbreviation>.h` – for services and type definitions of the CanDrv (e.g.: `Can_99_Ext1.h, Can_99_Ext2.h`)
- `CanTrcv_<vendorID>_<Vendor specific name><driver abbreviation>.h` – for services and type definitions of the CanTrcv (e.g.: `CanTrcv_99_Ext1.h`)
- `Dem.h` – for services of the DEM
- `Can_GeneralTypes.h` – for CanDrv generic definitions used by the CanIf
- `ComStack_Types.h` - for COM related type definitions
- `MemMap.h` – for accessing the module specific functionality provided by the BSW Memory Mapping⌋

(BSW00436)

Note: The following header files are indirectly included by `ComStack_Types.h:`

- `Std_Types.h` – for AUTOSAR standard types
- `Platform_Types.h` – for platform specific types
- `Compiler.h` – for compiler specific language extensions

[CANIF464] ⌈The CanIf may include following optional header file

- `Det.h` – for services of the DET⌋()

[CANIF208] ⌈The CanIf shall include the following header files `<Module>_CanIf.h` of those upper layer modules, from which declarations of only CanIf specific API services or type definitions are needed:

- `PduR_CanIf.h` – for services and callback declarations of the PduR
- `SchM_CanIf.h` –for services and callback declarations of the SchM

⌋(BSW00415)

[CANIF233] ⌈The CanIf shall include the following header files `<Module>_Cbk.h`, in which the callback functions called by the CanIf at the upper layers are declared:

- `CanSM_Cbk.h` – for callback declarations of the CanSm
- `CanNm_Cbk.h` – for callback declarations of the CanNm
- `CanTp_Cbk.h` – for callback declarations of the CanTp

- `EcuM_Cbk.h` – for callback declarations of the EcuM
- `<CDD>_Cbk.h` – for callback declarations of CDD; <CDD> is configurable via parameter CANIF_CDD_HEADERFILE (see CANIF671_Conf)

⌋()

[CANIF280] ⌈The CanIf shall include the following header files `<Module>_Cfg.h`, which contain the configuration data used by the CanIf:

- `Can_<vendorID>_<Vendor specific name><driver abbreviation>_Cfg.h` – for configuration data of the CanDrv (e.g.: `Can_99_Ext1_Cfg.h`)
- `CanTrcv_<Vendor Id>_<Vendor specific name><driver abbreviation>_Cfg.h` – for configuration data of the CanTrcv (e.g.: `CanTrcv_99_Ext1_Cfg.h`)
- `PduR_Cfg.h` – for PduR configuration data (e.g. PduR target PDU Ids)
- `CanNm_Cfg.h` – for CanNm configuration data (e.g. CanNm target PDU Ids)
- `CanTp_Cfg.h` – for CanTp configuration data (e.g. CanTp target PDU Ids)
- `Xcp_Cfg.h` - for XCP configuration data (e.g. XCP target PDU Ids)⌋()

[CANIF150] ⌈The CanIf shall include the file `Dem.h`.⌋()

By this way, reporting production errors as well as the required Event Id symbols are included. This specification defines the name of the Event Id symbols (see error table in chapter 7.27 Error classification), which are provided by XML to the DEM configuration tool. The DEM configuration tool assigns ECU dependent values to the Event Id symbols and publishes the symbols in `Dem_IntErrId.h`.

[CANIF278] ⌈The CanIf shall include the file `MemMap.h` in case the mapping of code and data to specific memory sections via memory mapping file is needed for CanIf implementation.⌋()

5.8 Version check

[CANIF021] ⌈The CanIf shall perform Inter Module Checks to avoid integration of incompatible files.
The imported included files shall be checked by preprocessing directives. ⌋(BSW004)

The following version numbers shall be verified (see CANIF728) :
- <MODULENAME>_AR_RELEASE_MAJOR_VERSION

- <MODULENAME>_AR_RELEASE_MINOR_VERSION
Where <MODULENAME> is the module abbreviation of the other (external) modules which provide header files included by the CanIf.

If the values are not identical to the expected values, an error shall be reported.

Hint: The CanIf files check the consistency between the header, C and configuration files during compilation according to BSW004 General Requirements on Basic Software Modules [3]. The CanIf's implementer shall avoid the integration of incompatible files. Minimum implementation is the version check of the header file.

6 Requirements traceability

Requirement	Description	Satisfied by
BSW00306	These requirements are not applicable to this specification.	CANIF999
BSW00307	These requirements are not applicable to this specification.	CANIF999
BSW00308	These requirements are not applicable to this specification.	CANIF999
BSW00309	These requirements are not applicable to this specification.	CANIF999
BSW00312	The CanIf shall protect preemptive events, which access shared resources, that could be changed d...	CANIF064
BSW00318	The CanIf shall provide the following version numbers with the following naming convention (see C...	CANIF728
BSW00321	The numbering of CANIF_SW_MAJOR_VERSION, CANIF_SW_NINOR_VERSION and CANIF_SW_PATCH_VERSION from ...	CANIF729
BSW00323	If parameter ConfigPtr of CanIf_Init() has an invalid value, the CanIf shall report development e...	CANIF302, CANIF311, CANIF774, CANIF313, CANIF656, CANIF319, CANIF320, CANIF652, CANIF325, CANIF326, CANIF331, CANIF336, CANIF341, CANIF346, CANIF657, CANIF658, CANIF352, CANIF353, CANIF538, CANIF648, CANIF364, CANIF650, CANIF537, CANIF649, CANIF535, CANIF536, CANIF398, CANIF404, CANIF410, CANIF416, CANIF417, CANIF418, CANIF419, CANIF424, CANIF828, CANIF429
BSW00325	If a target upper layer module was configured to be called with its providing receive indication ...	CANIF135
BSW00326	These requirements are not applicable to this specification.	CANIF999
BSW00328	These requirements are not applicable to this specification.	CANIF999
BSW00330	These requirements are not applicable to this specification.	CANIF999
BSW00334	These requirements are not applicable to this specification.	CANIF999
BSW00336	These requirements are not applicable to this specification.	CANIF999
BSW00338	If the CANIF_PUBLIC_DEV_ERROR_DETECT switch is enabled, API checking is enabled.	CANIF019
BSW00339	Production errors shall be reported to the Dem.	CANIF020
BSW00341	These requirements are not applicable to this specification.	CANIF999

BSW00342	Variant 3: Mix of pre compile-, link time and post build time parameters.	CANIF462
BSW00344	Variant 2: Mix of pre compile- and link time parameters.	CANIF461, CANIF462
BSW00347	If multiple CanDrvs are assigned to a CanIf, then that CanIf shall provide a separate set of call...	CANIF124
BSW00348		CANIF142
BSW00350	If the CANIF_PUBLIC_DEV_ERROR_DETECT switch is enabled, API checking is enabled.	CANIF019
BSW00353		CANIF142
BSW00358		CANIF001
BSW00361		CANIF142
BSW00369	The detection of development errors is configurable (ON / OFF) at pre-compile time.	CANIF018
BSW00373	These requirements are not applicable to this specification.	CANIF999
BSW00374	The CanIf shall provide a readable module vendor identification in its published parameters (see ...	CANIF726
BSW00376	These requirements are not applicable to this specification.	CANIF999
BSW00378	These requirements are not applicable to this specification.	CANIF999
BSW00379	The CanIf shall provide a module identifier in its published parameters (see CANIF725).	CANIF727
BSW00380	The code file structure shall not be defined within this specification completely.	CANIF374, CANIF376
BSW00381	The CanIf shall include necessary configuration data by the header files:	CANIF122
BSW00386	The detection of development errors is configurable (ON / OFF) at pre-compile time.	CANIF018, CANIF019, CANIF156
BSW004	The CanIf shall perform Inter Module Checks to avoid integration of incompatible files.	CANIF021
BSW00402	The standardized common published parameters as required by BSW00402 in the General Requirements ...	CANIF725
BSW00404	Variant 3: Mix of pre compile-, link time and post build time parameters.	CANIF462
BSW00405		CANIF001
BSW00407		CANIF158
BSW00409	Values for production code Event Ids are assigned externally by the configuration of the Dem.	CANIF153
BSW00411		CANIF158
BSW00412	The CanIf shall include necessary configuration data by the header files:	CANIF122
BSW00414		CANIF001
BSW00415	The CanIf shall include the following header files _CanIf.	CANIF208

BSW00416	These requirements are not applicable to this specification.	CANIF999
BSW00417	These requirements are not applicable to this specification.	CANIF999
BSW00419	The code-file structure shall include CanIf_Cfg.	CANIF376
BSW00423	These requirements are not applicable to this specification.	CANIF999
BSW00424	These requirements are not applicable to this specification.	CANIF999
BSW00425	These requirements are not applicable to this specification.	CANIF999
BSW00426	These requirements are not applicable to this specification.	CANIF999
BSW00427	These requirements are not applicable to this specification.	CANIF999
BSW00428	These requirements are not applicable to this specification.	CANIF999
BSW00429	These requirements are not applicable to this specification.	CANIF999
BSW00431	These requirements are not applicable to this specification.	CANIF999
BSW00432	These requirements are not applicable to this specification.	CANIF999
BSW00433	These requirements are not applicable to this specification.	CANIF999
BSW00434	These requirements are not applicable to this specification.	CANIF999
BSW00435	These requirements are not applicable to this specification.	CANIF999
BSW00436	The CanIf include the following header files .	CANIF463
BSW007	These requirements are not applicable to this specification.	CANIF999
BSW010	These requirements are not applicable to this specification.	CANIF999
BSW01001	The CanIf shall avoid direct access to hardware specific communication buffers and shall access i...	CANIF023
BSW01003		CANIF012
BSW01005	The CanIf shall accept all received L-PDUs (see CANIF390) with a DLC value equal or greater then ...	CANIF026
BSW01008		CANIF005
BSW01009		CANIF007
BSW01014	These requirements are not applicable to this specification.	CANIF999
BSW01015	The listed configuration items can be derived from a network description database, which is based...	CANIF104
BSW01018	If the CanIf has found the CanId of the received	CANIF030

	L-PDU in the list of receive CanIds for the HRH ...	
BSW01020	The CanIf shall support buffering of a CAN L-PDU handle for BasicCAN transmission in the CanIf, i...	CANIF063
BSW01021		CANIF001
BSW01022		CANIF001
BSW01024	These requirements are not applicable to this specification.	CANIF999
BSW01027		CANIF003
BSW01028		CANIF229
BSW01029		CANIF014
BSW01114	The CanIf shall protect access to transmit L-PDU buffers for all transmit L-PDUs by usage of crit...	CANIF033
BSW01125		CANIF194
BSW01126	If an L-PDU is requested to be transmitted via a PDU channel mode (refer to chapter 7.	CANIF382, CANIF381
BSW01129		CANIF194
BSW01130		CANIF202, CANIF230
BSW01131		CANIF230
BSW01136	(sources) shall be called during CanIf_CheckValidation(WakeupSource),...	CANIF179
BSW01139	These requirements are not applicable to this specification.	CANIF999
BSW01140	The CanIf shall accept and handle StandardCAN IDs and ExtendedCAN IDs on the same physical channe...	CANIF281
BSW01141	The CanIf shall set the 'identifier extension flag' (see [18]ISO11898 - Road vehicles - controlle...	CANIF243
BSW101		CANIF001
BSW159	These requirements are not applicable to this specification.	CANIF999
BSW164	These requirements are not applicable to this specification.	CANIF999
BSW167	These requirements are not applicable to this specification.	CANIF999
BSW168	These requirements are not applicable to this specification.	CANIF999
BSW170	These requirements are not applicable to this specification.	CANIF999
BSW172	These requirements are not applicable to this specification.	CANIF999

Document: General Requirements on Basic Software Modules [3]

[BSW00344] Reference to link-time configuration	CANIF461, CANIF462

[BSW00404] Reference to post build time configuration	CANIF462
[BSW00405] Reference to multiple configuration sets	CANIF001, chapter 8.2.1 CanIf_ConfigType
[BSW00345] Pre-Build Configuration	Fulfilled by configuration parameter definitions in chapter 10. The configuration parameters are described in a general way.
[BSW159] Tool-based configuration	Not applicable (assigned to configuration tool)
[BSW167] Static configuration checking	Not applicable (assigned to configuration tool)
[BSW171] Configurability of optional functionality	Fulfilled by configuration parameter definitions in chapter 10. The configuration parameters are described in a general way.
[BSW170] Data for reconfiguration of SW-components	Not applicable (no interface to AUTOSAR SW Components)
[BSW00380] Separate C-Files for configuration parameters	CANIF374, CANIF376
[BSW00419] Separate C-Files for pre-compile time configuration parameters	CANIF376
[BSW00381] Separate configuration header file for pre-compile time parameters	CANIF122
[BSW00412] Separate H-File for configuration parameters	CANIF122
[BSW00383] List dependencies of configuration files	Subchapter 5.7.2 Header file structure
[BSW00384] List dependencies to other modules	Chapter 5 Dependencies to other modules, subchapter 5.4 Lower layers: CAN Driver
[BSW00387] Specify the configuration class of call-out function	Fulfilled by API definitions in chapter 8.
[BSW00388] Introduce containers	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00389] Containers shall have names	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00390] Parameter content shall be unique within the module	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00391] Parameter shall have unique names	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00392] Parameters shall have a type	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00393] Parameters shall have a range	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00394] Specify the scope of the parameters	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00395] List the required parameters (per parameter)	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00396] Configuration classes	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00397] Pre-compile-time parameters	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00398] Link-time parameters	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00399] Loadable Post-build time parameters	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00400] Selectable Post-build time parameters	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00402] Published information	CANIF725

[BSW00375] Notification of wake-up reason	CANIF013
[BSW101] Initialization interface	CANIF001
[BSW00416] Sequence of Initialization	Not applicable (no initialization dependencies for this module)
[BSW00406] Check module initialization	Fulfilled by API definitions in chapter 8.
[BSW168] Diagnostic Interface of SW components	Not applicable (this module does not support a special diagnostic interface)
[BSW00407] Function to read out published parameters	CANIF158
[BSW00423] Usage of SW-C template to describe BSW modules with AUTOSAR Interfaces	Not applicable (this module does not provide an AUTOSAR interface)
[BSW00424] BSW main processing function task allocation	Not applicable (requirement on system design, not on a single module)
[BSW00425] Trigger conditions for schedulable objects	Not applicable (requirement on system configuration, not on a single module)
[BSW00426] Exclusive areas in BSW modules	Not applicable (no exclusive areas specified for this module)
[BSW00427] ISR description for BSW modules	Not applicable (this module does not provide any ISRs)
[BSW00428] Execution order dependencies of main processing functions	Not applicable (No scheduled API)
[BSW00429] Restricted BSW OS functionality access	Not applicable (this module doesn't use any OS objects or services)
[BSW00431] The BSW Scheduler module implements task bodies	Not applicable (No scheduled API)
[BSW00432] Modules should have separate main processing functions for read/receive and write/transmit data path	Not applicable (requirement on the CAN Driver module)
[BSW00433] Calling of main processing functions	Not applicable (requirement on the BSW scheduler module)
[BSW00434] The Schedule Module shall provide an API for exclusive areas	Not applicable (requirement on the BSW scheduler module)
[BSW00336] Shutdown interface	Not applicable (architecture decision)
[BSW00337] Classification of errors	Table in section 7.27 Error classification
[BSW00338] Detection and Reporting of development errors	CANIF019
[BSW00369] Do not return development error codes via API	CANIF018
[BSW00339] Reporting of production relevant error status	CANIF020
[BSW00417] Reporting of Error Events by Non-Basic Software	Not applicable (this is a basic software module)
[BSW00323] API parameter checking	CANIF302, CANIF311, CANIF313, CANIF319, CANIF320, CANIF325, CANIF326, CANIF331, CANIF336, CANIF341, CANIF346, CANIF352, CANIF353, CANIF364, CANIF398, CANIF404, CANIF410, CANIF416, CANIF417, CANIF418, CANIF419, CANIF424, CANIF429, CANIF535, CANIF536, CANIF537, CANIF538, CANIF648, CANIF649, CANIF650, CANIF652, CANIF656, CANIF657, CANIF658
[BSW004] Version check	CANIF021
[BSW00409] Header files for production code	CANIF153

error IDs	
[BSW00385] List possible error notifications	Table in section 7.27 Error classification
[BSW00386] Configuration for detecting an error	CANIF018, CANIF019, CANIF156
[BSW161] Microcontroller abstraction	chapter 5.6 Configuration
[BSW162] ECU layout abstraction	chapter 5.6 Configuration
[BSW005] No hard coded horizontal interfaces within MCAL	Subchapter 5.7.2Header file structure
[BSW00415] User dependent include files	CANIF208
[BSW164] Implementation of interrupt service routines	Not applicable
[BSW00325] Runtime of interrupt service routines	CANIF135 The runtime is not totally under control of the CAN Interface, because they are called to the upper layers.
[BSW00326] Transition from ISRs to OS tasks	Not applicable (When a transition from ISR to OS task is done, it will be defined in COM Stack SWS)
[BSW00342] Usage of source code and object code	CANIF462CANIF228 (post build configuration)
[BSW00343] Specification and configuration of time	Not applicable (no internal scheduling policy)
[BSW160] Human-readable configuration data	Fulfilled by configuration parameter definitions in chapter 10. The configuration parameters are described in a general way.
[BSW007] HIS MISRA C	Not applicable (requirement on implementation, not on specification)
[BSW00300] Module naming convention	Fulfilled by API definitions in chapter 8.
[BSW00413] Accessing instances of BSW modules	Fulfilled by API definitions in chapter 8.
[BSW00347] Naming separation of different instances of BSW drivers	CANIF124
[BSW00305] Self-defined data types naming convention	Fulfilled by type definitions in chapter 8.2.
[BSW00307] Global variables naming convention	Not applicable (requirement on implementation, not on specification)
[BSW00310] API naming convention	Fulfilled by API definitions in chapter 8.
[BSW00373] Main processing function naming convention	Not applicable (No scheduled API)
[BSW00327] Error values naming convention	Table in section 7.27 Error classification
[BSW00335] Status values naming convention	Subchapter 8.2.3 CanIf_PduGetModeType, subchapter 8.2.4 CanIf_PduSetModeType, subchapter 8.2.5 CanIf_NotifStatusType
[BSW00350] Development error detection keyword	CANIF019
[BSW00408] Configuration parameter naming convention	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00410] Compiler switches shall have defined values	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00411] Get version info keyword	CANIF158
[BSW00346] Basic set of module files	Subchapter 5.7.2 Header file structure
[BSW158] Separation of configuration from implementation	Subchapter 5.7.2 Header file structure
[BSW00314] Separation of interrupt frames and service routines	Not applicable (this module does not provide any ISRs)
[BSW00370] Separation of call-out interface from	Subchapter 5.7.2 Header file structure

API	
[BSW00435] Module Header File Structure for the Basic Software Scheduler	Not applicable
[BSW00436] Module Header File Structure for the Basic Software Memory Mapping	CANIF463
[BSW00348] Standard type header	CANIF142
[BSW00353] Platform specific type header	CANIF142 (automatically included with Standard types)
[BSW00361] Compiler specific language extension header	CANIF142 (automatically included with Standard types)
[BSW00301] Limit imported information	Subchapter 5.7.2 Header file structure
[BSW00302] Limit exported information	
[BSW00328] Avoid duplication of code	Not applicable (requirement on implementation, not on specification)
[BSW00312] Shared code shall be reentrant	CANIF064
[BSW006] Platform independency	Fulfilled by API definitions in chapter 8.3
[BSW00357] Standard API return type	Fulfilled by API definitions in chapter 8.3.
[BSW00377] Module Specific API return type	Subchapter 8.2.3 CanIf_PduGetModeType, subchapter 8.2.4 CanIf_PduSetModeType, subchapter 8.2.5 CanIf_NotifStatusType
[BSW00304] AUTOSAR integer data types	Fulfilled by type and API definitions in chapter 8.1 and 8.2
[BSW00355] Do not redefine AUTOSAR integer data types	Fulfilled by type and API definitions in chapter 8.1 and 8.2
[BSW00378] AUTOSAR Boolean type	Not applicable (no Boolean types used)
[BSW00306] Avoid direct use of compiler and platform specific keywords	Not applicable (requirement on implementation, not on specification)
[BSW00308] Definition of global data	Not applicable (requirement on implementation, not on specification)
[BSW00309] Global data with read-only constraint	Not applicable (requirement on implementation, not on specification)
[BSW00371] Do not pass function pointers via API	Fulfilled by API definitions in chapter 8.3
[BSW00358] Return type of init() functions	CANIF001
[BSW00414] Parameter of init function	CANIF001
[BSW00376] Return type and parameters of main processing functions	Not applicable
[BSW00359] Return type of call-out functions	Fulfilled by call-out APIs in chapter 8.4.
[BSW00360] Parameters of call-out functions	Fulfilled by call-out APIs in chapter 8.4.
[BSW00329] Avoidance of generic interfaces	No generic interface used The content of functions might be configuration dependent. The scope of function is always defined
[BSW00330] Usage of macros instead of functions	Not applicable (requirement on implementation, not on specification)
[BSW00331] Separation of error and status values	section 7.27 Error classification, section 8.2.2 CanIf_ControllerModeType, section 8.2.5 CanIf_NotifStatusType
[BSW009] Module User Documentation	Fulfilled by the complete documentation.
[BSW00401] Documentation of multiple instances of configuration parameters	Fulfilled by configuration parameter definitions in chapter 10.
[BSW172] Compatibility and documentation of scheduling strategy	Not applicable (no internal scheduling policy)

BSW010] Memory resource documentation	Not applicable (requirement on implementation, not on specification)
[BSW00333] Documentation of callback function context	Fulfilled by callback functions in chapter 8.4.
[BSW00374] Module vendor identification	CANIF726
[BSW00379] Module identification	CANIF727
[BSW003] Version identification	CANIF021
[BSW00318] Format of module version	CANIF728
[BSW00321] Enumeration of module version numbers	CANIF729
[BSW00341] Microcontroller compatibility documentation	Not applicable (no microcontroller dependent module)
[BSW00334] Provision of XML file	Not applicable (requirement on implementation, not on specification)

Document: Requirements on CAN [4]

Requirement	***Satisfied by***
[BSW01033] Basic Software General Requirements	Fulfilled by this chapter.
[BSW01125] Data throughput read direction	CANIF194
[BSW01126] Data throughput write direction	CANIF381, CANIF382
[BSW01139] CAN Controller specific Initialization	Not applicable
[BSW01129] Receive Data Interface for CAN Interface and CAN Driver Module	Subchapter 7.16 Read received data, subchapter 8.3.6 CanIf_ReadRxPduData, CANIF194
[BSW01121] Interfaces of the CAN Interface module	Subchapter 5.4 Lower layers: CAN Driver, subchapter 5.5 Lower layers: CAN Transceiver Driver
[BSW01014] Network configuration abstraction	Not applicable
[BSW01001] HW independence	CANIF023
[BSW01015] Network Database Information Import	CANIF104
[BSW01016] Interface to CAN Driver configuration	Chapter 10.2
[BSW01018] Software Filter	CANIF030
[BSW01019] DLC Check configuration	chapter 10.2
[BSW01020] Tx Buffer configuration	CANIF063
[BSW01021] CAN Interface Module Power-On Initialization	CANIF001
[BSW01022] Dynamic selection of static configuration sets	CANIF001
[BSW01023] Power-On Initialization Sequence	Chapter 7.8
[BSW01002] Rx PDU dispatching	CANIF024
[BSW01003] Reception indication dispatcher	CANIF012
[BSW01114] Data Consistency of transmit L-PDUs	CANIF033
[BSW01004] Software Filtering for L-PDU reception	Subchapter 7.21
[BSW01005] DLC check for L-PDU reception	CANIF026
[BSW01006] Rx L-PDU enable/disable	CANIF096
[BSW01007] Tx L-PDU dispatching	CANIF024
[BSW01008] Transmission request service	CANIF005
[BSW01009] Transmission confirmation service	CANIF007
[BSW01011] Tx buffering	CANIF068
[BSW01013] Tx L-PDU enable/disable service	CANIF096

[BSW01027] CAN controller Mode Select service	CANIF003
[BSW01028] CAN controller State Service	CANIF229
[BSW01032] Wake-up Notification	CANIF013
[BSW01061] Dynamic Tx Handles	Chapter 7.4
[BSW01024] DLC Error Notification	Not applicable
[BSW01029] Bus-off notification	CANIF014
[BSW01130] Read Status Interface of CAN Interface	CANIF202, CANIF230
[BSW01131] Mixed mode of notification and polling mechanism	CANIF230
[BSW01136] Notification of first received CAN message	CANIF179
[BSW01129] Receive Data Interface for CAN Interface	CANIF194
[BSW01140] Support of Standard and Extended Identifiers	CANIF281
[BSW01141] Support of both Standard and Extended Identifiers on one network (optional feature)	CANIF243, CANIF261

7 Functional specification

7.1 General functionality

The services of the CanIf can be divided into the following main groups:

- Initialization
- Transmit request services
- Transmit confirmation services
- Reception indication services
- Controller mode control services
- PDU mode control services

Possible applications of the CanIf:

i. Interrupt mode
The CanDrv processes interrupts triggered by the CAN controller. The CanIf, which is event based, is notified when the event occurs. In this case the relevant CanIf services is called within the corresponding ISRs in the CanDrv.

ii. Polling mode
The CanDrv is triggered by the SchM and performs subsequent processes (polling mode). In this case `Can_MainFunction_<Write/Read/BusOff/Wakeup/Transceiver>()` must be called periodically within a defined time interval. The CanIf is notified by the CanDrv about events (reception, transmission, BusOff, TxCancellation, Timeout), that occurred in one of the CAN controllers, equally to the interrupt driven operation. The CanDrv is responsible for the update of the corresponding information which belongs to the occurred event in the CAN controller, for example reception of an L-PDU.

iii. Mixed mode: interrupt and polling driven CanDrv
The functionality can be divided between interrupt driven and polling driven operation mode depending on the used CAN controllers.
Examples: Polling driven FullCAN reception and interrupt driven BasicCAN reception, polling driven transmit and interrupt driven reception, etc.

This specification describes a unique interface, which is valid for all three types of operation modes. Summarized the CanIf works in the same way, either if any events are processed on interrupt, task level or mixed. The only difference is the call context and probably the way of interruption of the notifications: pre-emptive or co-operative. All services are performed in accordance with the configuration.

The following paragraphs describe the functionality of the CanIf.

7.2 Hardware object handles

Hardware object handles (HOH) for transmission (HTH) as well as for reception (HRH) represent an abstract reference to a CAN mailbox structure, that contains CAN related parameters such as CanId, DLC and data. Based on the CAN hardware

buffer abstraction each hardware object is referenced in the CanIf independent of the CAN hardware buffer layout. The HOH is used as a parameter in the calls of the CanDrv's interface services and is provided by the CanDrv's configuration and used by the CanDrv as identifier for communication buffers of the CAN mailbox.
The CanIf acts only as user of the Hardware object handle, but does not interpret it on the basis of hardware specific information. The CanIf therefore remains independent of hardware.

[CANIF023] ⌈The CanIf shall avoid direct access to hardware specific communication buffers and shall access it exclusively via the CanDrv interface services.⌋(BSW01001)

Rationale for CANIF023: The CanIf remains independent of hardware, because the CanDrv interfaces are called with HOH parameters, which abstract from the concrete CAN hardware buffer properties.

Each CAN controller can provide multiple CAN transmit hardware objects in the CAN mailbox. These can be logically linked to one entire pool of hardware objects (multiplexed hardware objects) and thus addressed by one HTH.

CANIF662: The CanIf shall use two types of HOHs to enable access to the CanDrv:

- Hardware Transmit Handle (HTH) and
- Hardware Receive Handle (HRH).

[CANIF291] ⌈Definition of HRH: The HRH shall be a handle referencing a logical hardware receive object of the CAN controller mailbox. ⌋()

[CANIF665] ⌈The HRH shall enable the CanIf to use BasicCAN or a FullCAN reception method of the referenced reception unit and to indicate a received L-PDU to a target upper layer module.⌋()

[CANIF663] ⌈If the HRH references a reception unit configured for BasicCAN transmission, software filtering shall be enabled in the CanIf. ⌋()

[CANIF465] ⌈Each CanRxPduId shall be assigned to one or multiple HRHs. Thus the assignment of single CanIds to multiple HRHs is possible.⌋()

[CANIF664] ⌈If multiple HRHs are used, each HRH shall belong at least to a single or fixed group of Rx L-PDU handles (CanRxPduIds).⌋()

The HRH can be configured to receive

- one single CanId (FullCAN)
- a group of single CanIds (BasicCAN)
- a range/area of CanIds (BasicCAN) or

- all CanIds.

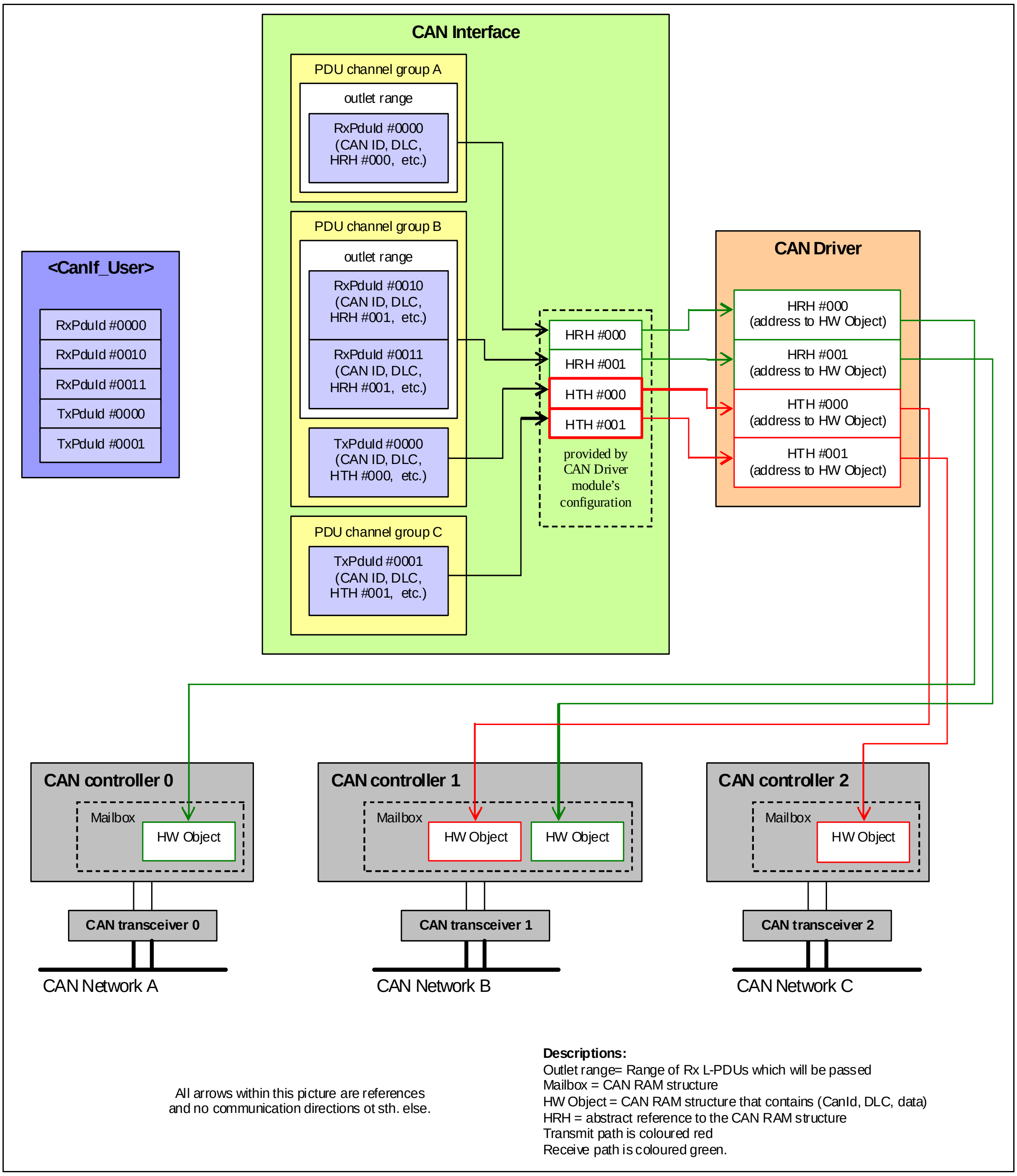


Figure 4: Mapping between PDU Ids and HW object handles

[CANIF292] ⌈Definition of HTH: The HTH shall be a handle referencing a logical hardware transmit object of the CAN controller mailbox.⌋()

[CANIF666] ⌈The HTH shall enable the CanIf to use BasicCAN or a FullCAN transmission method of the referenced transmission unit and to confirm a transmitted L-PDU to a target upper layer module. ⌋()

[CANIF466] ⌈Each CanIf Tx L-PDU shall statically be assigned to one CanIfTxbuffer (see CANIF832_Conf) configuration container at configuration time (see CANIF831_Conf).⌋()
Rationale for CANIF466: CanIf Tx L-PDUs do not refer HTHs, but CanIfTxBuffer, which in turn do refer HTHs.

[CANIF667] ⌈If multiple HTHs are used, each HTH shall belong to a single or fixed group of Tx L-PDU handles (CanTxPduIds). ⌋()

[CANIF115] ⌈The CanIf shall be able to use all HRHs and HTHs of one CanDrv as common, single numbering area starting with zero.⌋()

The dedicated HRH and HTHs are derived from the configuration set of the CanDrv. The definition of HTH/HRH inside the numbering area and hardware objects is up to the CanDrv. It has to be ensured by configuration, that no overlapping of several numbering areas of multiple CanDrvs is allowed.

7.3 Static CAN L-PDU handles

The CanIf offers general access to the CAN L-PDU related data for upper layers. The L-PDU handle facilitates this access. The L-PDU handle refers to data structures, which consists of CanIf specific and CAN PCI specific attributes describing the L-PDU. Attributes of the following table are represented as configuration parameters and are specified in chapter 10:

CAN Interface specific attributes	*CAN Protocol Control Information (PCI)*
Method of SW filtering CANIF_PRIVATE_SOFTWARE_FILTER_TYPE (see CANIF619_Conf)	CAN Identifier (ID) CANIF_TXPDU_CANID (see CANIF592_Conf), range of CanIds per PDU (see CANIF743_Conf)
Direction of L-PDU (Tx, Rx) CANIF_TXPDU_ID (seeCANIF591_Conf), CANIF_RXPDU_ID (seeCANIF597_Conf)	Type of CAN Identifier (StandardCAN, ExtendedCAN) referenced from CanDrv via CANIF_HTH_ID_SYMREF (see CANIF627_Conf), CANIF_HRH_ID_SYMREF (see CANIF634_Conf)
CAN Hardware Unit (CANIF_PUBLIC_NUMBER_OF_CAN_HW_UNITS (see CANIF615_Conf)	Data Length Code (DLC) CANIF_TXPDU_DLC (see CANIF594_Conf), CANIF_RXPDU_DLC (seeCANIF599_Conf)
HTH/HRH of the CAN controller	Reference to the data (SDU) (see [8]Specification of CAN Driver)
Target ID for the corresponding upper layer CANIF_TXPDU_USERTXCONFIRMATION (see CANIF527_Conf), CANIF_RXPDU_USERRXINDICATION_UL (seeCANIF529_Conf)	
Type of transmit L-PDU handle (static, dynamic) CANIF_TXPDU _TYPE (see CANIF593_Conf)	
Type of Tx/Rx L-PDU (FullCAN, BasicCAN) CANIF_HTH_ID_SYMREF (see CANIF627_Conf),	

CANIF_HRH_ID_SYMREF (see CANIF634_Conf)	

Table 1 Attributes used in CAN Interface

[CANIF046] ⌈The CanIf shall assign each L-PDU handle to one CAN controller only. Thus, the assignment of single L-PDU handles to more than one CAN controller is prohibited. ⌋()

Rationale for CANIF046: This relation is used in order to ensure correct L-PDU dispatching at transmission confirmation and reception indication events. In this manner the CanIf is able to identify the CAN controller module from the L-PDU handle.

The CanIf supports activation and deactivation of all L-PDUs belonging to one CAN controller for transmission as well as for reception (see chapter 7.20.2 PDU channel modes ,see `CanIf_SetPduMode()`, CANIF008). For L-PDU mode control refer to section [7.20 PDU channel mode control].

Each L-PDU handle is associated with an upper layer module in order to ensure correct dispatching during reception, transmission confirmation and data access. Each upper layer module can use the L-PDU handles to serve different CAN controllers simultaneously.

According to the PDU architecture defined for the entire AUTOSAR communication stack (see [2] Layered Software Architecture), the usage of L-PDUs is split in two different ways:

For transmission request and transmission/reception polling API the upper layer module uses the CAN L-PDU Id defined by the CanIf as parameter.
For all callback APIs, which are invoked by the CanIf at upper layer modules, the CanIf passes the target PduId defined by each upper layer module as parameter.

The principle is that the caller must use the defined target PDU Id of the callee.

If power on initialization is not performed and upper layer performs transmit requests to CanIf, no L-PDUs are transmitted to lower layer and DET shall be invoked. Thus, no un-initialized data can be transmitted on the network. Behavior of PDU transmitting function is specified in detail in chapter [8.3.4 CanIf_Transmit].

7.4 Dynamic CAN transmit L-PDU handles

Definition of dynamic transmit L-PDUs: L-PDUs handle which allows reconfiguration of the CanId of the corresponding used L-PDU handle during runtime.

The usage of all other L-PDU elements are equal to normal static transmit L-PDUs:

- The transmit confirmation notification `CANIF_TXPDU_USERTXCONFIRMATION_UL` (see CANIF527_Conf) cannot be reconfigured as it belongs to the L-PDU handle.
- The data length code (DLC) and the pointer to the data buffer are both determined by the upper layer module at call of `CanIf_Transmit()`.

The function `CanIf_SetDynamicTxId()` reconfigures the CanId of a L-PDU (see CANIF189).

[CANIF188] ⌈ The CanIf shall process the ‘identifier extension flag’ (see [18]ISO11898 – Road vehicles - controller area network (CAN)) to determine the kind of CanId and thus how the dynamic transmit L-PDU shall be transmitted.⌋()

[CANIF673] ⌈The CanIf shall guarantee data consistency of the CanId in case of running function `CanIf_SetDynamicTxId()`. This service may be interrupted by a pre-emptive call of `CanIf_Transmit()` affecting the same L-PDU handle, see CANIF064. ⌋()

Note: `CanIf_Init()` initializes the CanIds of the dynamic transmit L-PDUs (see CANIF085).

7.5 Physical channel view

A physical channel is linked with one CAN controller and one CAN transceiver, whereas one or multiple physical channels may be connected to a single network.

The CanIf provides services to control all CAN devices like CAN Controllers and CAN Transceivers of all supported ECU's CAN channels. Those APIs are used by the CanSm to provide a network view to the ComM (see [11]Specification of CAN State Manager) used to perform wake up and sleep request for all physical channels connected to a single network.

The CanIf passes status information provided by the CanDrv and CanTrcv separately for each physical channel as status information for the CanSm (`<User_ControllerBusOff>()`, refer to CANIF014).

[CANIF653] ⌈The CanIf shall provide a `ControllerId`, which abstracts from the different `Controllers` of the different CanDrv instances. The range of the `ControllerIds` within the CanIf shall start with ‘0’. It shall be configurable via `CANIF_CTRL_ID` (see CANIF647_Conf).⌋()

Example:

CanIf	**CanDrv A**	**CanDrv B**
ControllerId 0	Controller 0	
ControllerId 1	Controller 1	
ControllerId 2		Controller 0

[CANIF655] ⌈The CanIf shall provide a `TransceiverId`, which abstracts from the different `Transceivers` of the different CanTrcv instances. The range of the `TransceiverIds` within the CanIf shall start with ‘0’. It shall be configurable via `CANIF_TRCV_ID` (see CANIF654_Conf).⌋()

Example:

CanIf	CanTrcv A	CanTrcv B
TransceiverId 0	Transceiver 0	
TransceiverId 1	Transceiver 1	
TransceiverId 2		Transceiver 0

During the notification process the CanIf maps the original CAN controller or CAN transceiver parameter from the Driver module to the CanSm. This mapping is done as the referenced CAN controller or CAN transceiver parameters are configured with the abstracted CanIf parameters `ControllerId` or `TransceiverId`.

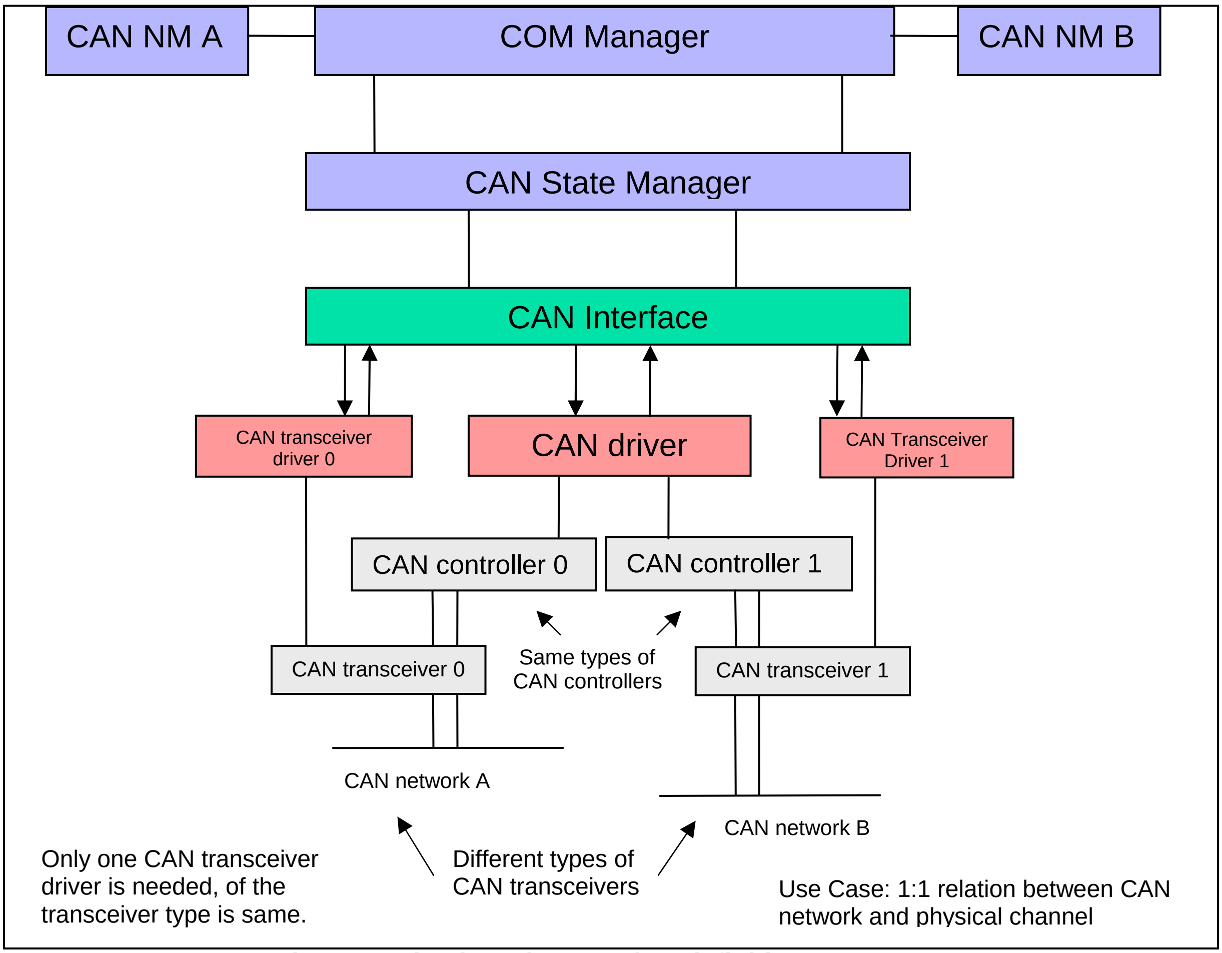


Figure 5: Physical channel view definition example A

The CanIf supports multiple physical CAN channels. These have to be distinguished by the CanSm for network control. The CanIf API provides request and read control for multiple underlying physical CAN channels.

Moreover the CanIf does not distinguish between dedicated types of CAN physical layers (i.e. Low-Speed CAN or High-Speed CAN), to which one or multiple CAN controllers are connected.

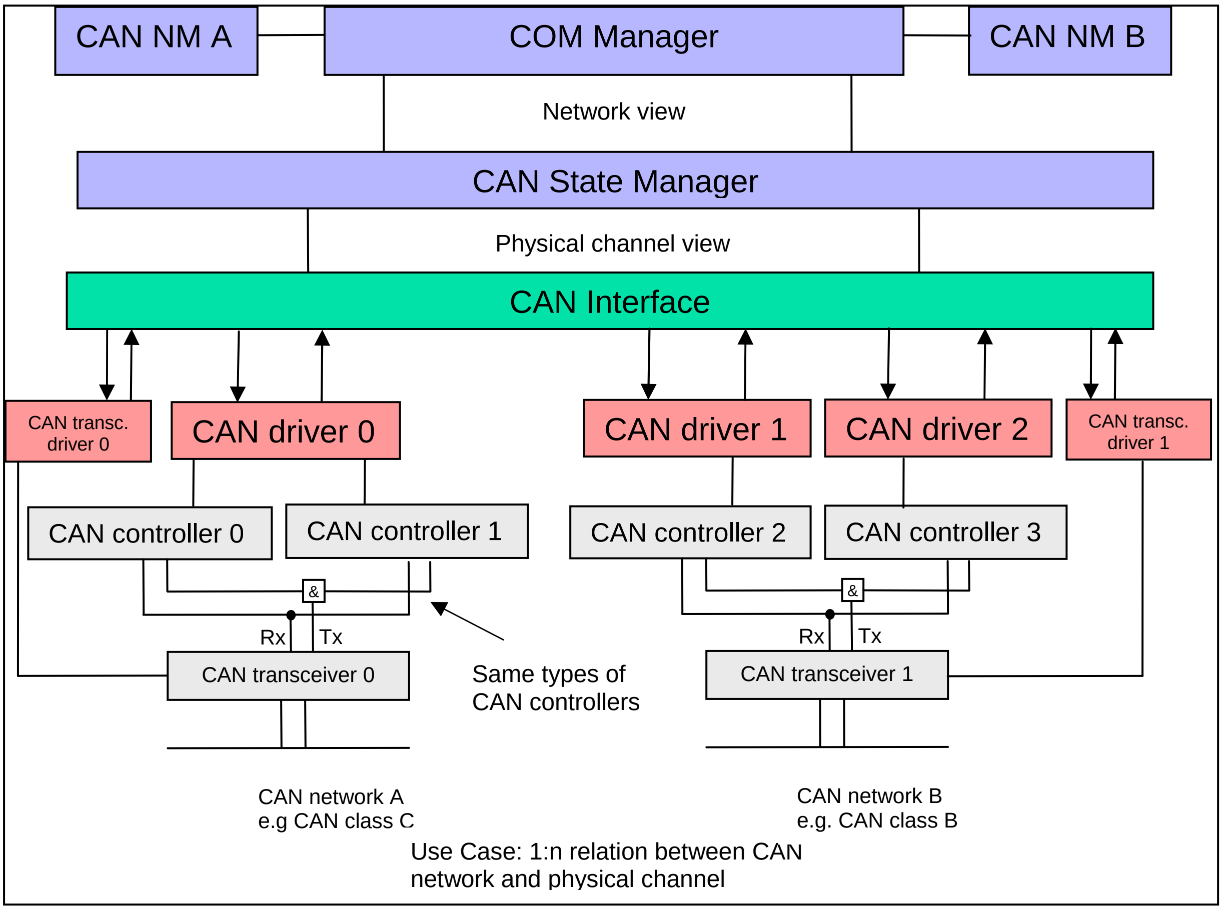


Figure 6: Physical channel view definition example B

7.6 CAN hardware unit

The CAN hardware unit combines one or multiple CAN controller modules of the same type, which may be located on-chip or as external standalone devices. Each CAN hardware unit is served by the corresponding CAN Driver module.

If different types of CAN controllers are used, also different types of CAN Driver modules have to be applied with a unified API to the CAN Interface module. The CAN Interface module collects information about number and types of CAN controller modules and their hardware objects in its mapping tables at configuration time. This allows transparent and hardware independent access to the CAN controllers from upper layer modules using HOHs (refer to [7.2 Hardware object handles] and [7.25 Multiple CAN Driver support]).

The following figure shows a CAN hardware unit consisting of two CAN controllers of the same type connected to two physical channels:

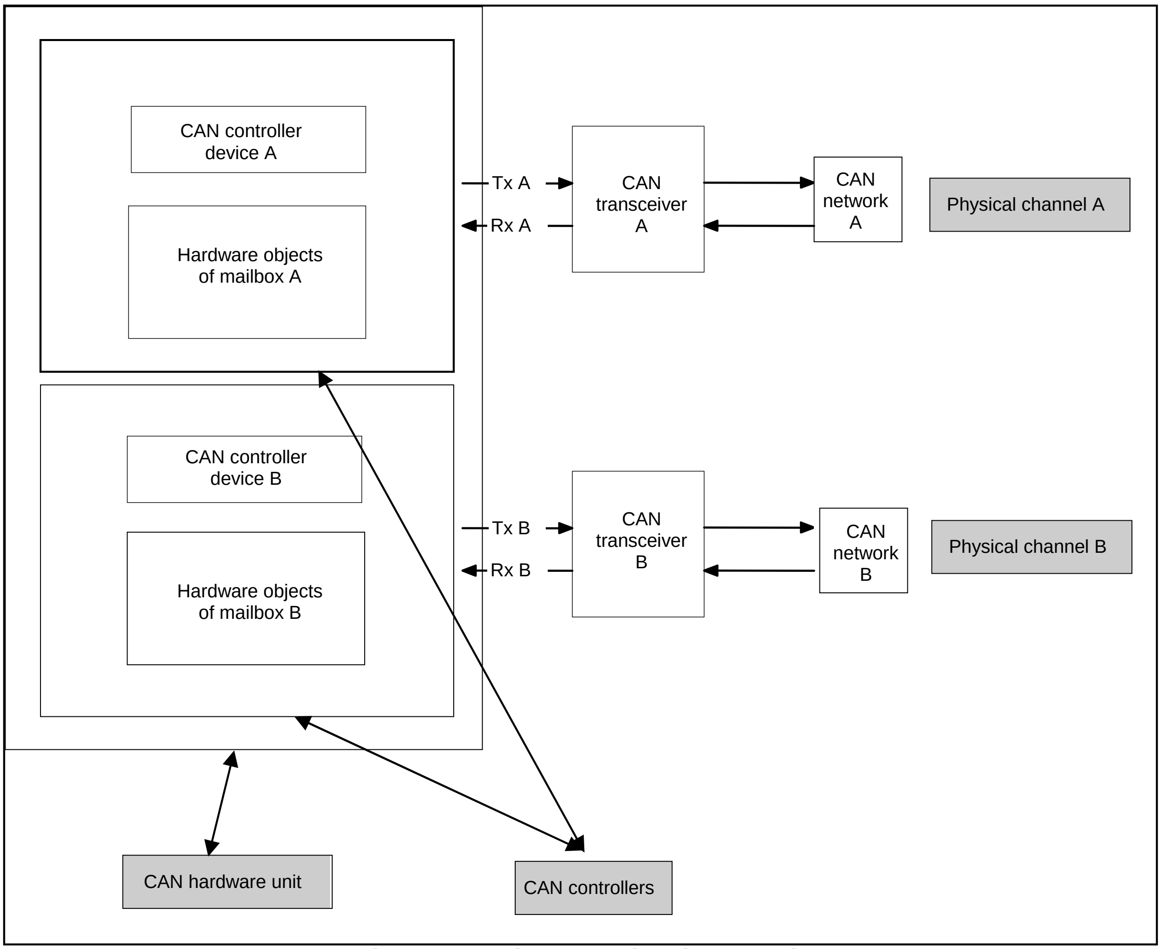


Figure 7 Typical CAN hardware unit

7.7 BasicCAN and FullCAN reception

The CanIf distinguishes between BasicCAN and FullCAN handling for activation of software acceptance filtering.

A CAN mailbox (hardware object) for FullCAN operation only enables transmission or reception of single CanIds. Accordingly, BasicCAN operation of one hardware object enables to transmit or receive a range of CanIds.

A hardware receive object for configured BasicCAN reception is able to receive a range of CanIds, which pass its hardware acceptance filter.
This range may exceed the list of predefined Rx L-PDUs to be received by this HRH. Therefore the CanIf subsequently shall execute software filtering to pass only the predefined list of Rx L-PDUs to the corresponding upper layer modules. For more details please refer to [7.21Software receive filter].

[CANIF467] ⌈The CanIf shall configure and store an order on HTHs and HRHs for all HOHs derived from the configuration containers CanIfHthCfg (see CANIF258_Conf) and CanIfHrhCfg (see CANIF259_Conf).⌋()

[CANIF468] ⌈The CanIf shall reference a hardware acceptance filter for each HOH derived from the configuration parameters CANIF_HTH_Id_SYMREF (see CANIF627_Conf) and CANIF_HRH_ID_SYMREF (see CANIF634_Conf). ⌋()

The main difference between BasicCAN and FullCAN operation is in the need of a software acceptance filtering mechanism (see chapter 7.21 Software receive filter).

[CANIF469] ⌈The CanIf shall give the possibility to configure and store a software acceptance filter for each HRH of type BasicCAN configured by parameter CANIF_HRH_SOFTWARE_FILTER (see CANIF632_Conf).⌋()

[CANIF211] ⌈The CanIf shall execute the software acceptance filter from CANIF469 for the HRH passed by callback function CanIf_RxIndication().⌋()

BasicCAN and FullCAN objects may coexist in a single configuration setup. Multiple BasicCAN and FullCAN receive objects can be used, if provided by the underlying CAN controllers.

Basically the CanIf supports reception either of StandardCAN IDs or ExtendedCAN IDs on one physical CAN channel by the parameters CANIF_CANTXPDUID_CANIDTYPE (see CANIF590_Conf) and CANIF_CANRXPDUID_CANIDTYPE (see CANIF596_Conf).

[CANIF281] ⌈The CanIf shall accept and handle StandardCAN IDs and ExtendedCAN IDs on the same physical channel (=mixed mode operation).⌋ (BSW01140)

In a mixed mode operation StandardCAN IDs and ExtendedCAN IDs can be used mixed at the same time on the same CAN network. Mixed mode operation can be accomplished, if the BasicCAN/FullCAN hardware objects have been configured separately for either StandardCAN or ExtendedCAN operation using configuration parameters CANIF_CANTXPDUID_CANIDTYPE (see CANIF590_Conf) and CANIF_CANRXPDUID_CANIDTYPE (see CANIF596_Conf). In case of mixed mode operation the software acceptance filter algorithm (see 7.21 Software receive filter) must be able to deal with both type of CanIds.

CANIF281 is an optional feature. This feature can be realized by different variants of implementations, no configuration options are available.

7.8 Initialization

The EcuM calls the CanIf's function CanIf_Init() for initialization of the entire CanIf (see CANIF001). All global variables and data structures are initialized including flags and buffers during the initialization process. The EcuM executes initialization of CanDrvs and CanTrcvs separately by call of their corresponding initialization services (refer to [8] Specification of CAN Driver and [9]Specification of CAN Transceiver Driver).

The EcuM is responsible to ensure, that Initialization processes shall only take place, if all CCMSMs (see chapter 7.19.2 CAN Controller operation modes) for the corresponding CAN controllers equal `CANIF_CS_UNINIT` or `CANIF_CS_STOPPED`. `CANIF_CS_UNINIT` mode is left only, if once global initialization after power-on reset has been requested (see [15]Specification of ECU State Manager).

The CanIf expects that the CAN controller remains in STOPPED mode like after power-on reset after the initialization process has been completed. In this mode the CanIf and CanDrv are neither able to transmit nor receive CAN L-PDUs (see CANIF001).

If re-initialization of the entire CAN modules during runtime is required, the EcuM shall invoke the CanSm (see [11]Specification of CAN State Manager) to initiate the required state transitions of the CAN controller by call of CAN Interface module's API service `CanIf_SetControllerMode()`. The CanIf maps the calls from CanSm to calls of the respective CanDrvs (see chapter 8.3).

7.9 Transmit request

The CanIf's transmit request function `CanIf_Transmit()` (CANIF005) is a common interface for upper layers to transmit PDUs on the CAN network. The upper communication layer modules initiate the transmission only via the CAN Interface module's services without direct access to the CanDrv. The initiated transmit request is successfully completed, if the CanDrv could write the L-PDU data into the CAN hardware transmit object.

Upper layer modules use the API service `CanIf_Transmit()` to initiate a transmit request (refer to chapter [8.3.4 CanIf_Transmit].

The CanIf performs following actions for L-PDU transmission at call of the service `CanIf_Transmit()`:

- Check, initialization status of the CanIf
- Identify CanDrv (only if multiple CanDrvs are used)
- Determine HTH for access to the CAN hardware transmit object
- Call `Can_Write()` of the CanDrv

The transmission is successfully completed, if the transmit request service `CanIf_Transmit()` returns `E_OK`.

[CANIF382] ⌈If an L-PDU is requested to be transmitted via a PDU channel mode (refer to chapter 7.20.2 PDU channel modes), which equals CANIF_OFFLINE, the CanIf shall report the development error code `CANIF_E_STOPPED` to the `Det_ReportError` service of the DET and `CanIf_Tranmsit()` shall return `E_NOT_OK`.⌋(BSW01126)

[CANIF723]⌈If an L-PDU is requested to be transmitted via a CAN controller, whose CCMSM (see chapter 7.19) equals CANIF_CS_STOPPED, the CanIf shall report the development error code CANIF_E_STOPPED to the Det_ReportError service of the DET and CanIf_Transmit() shall return E_NOT_OK.⌋()

If the call of Can_Write()returns with CAN_BUSY, please refer to [7.12 Transmit buffering] for further details.

7.10 Transmit data flow

The transmit request service CanIf_Transmit() is based on L-PDU handles . The access to the L-PDU specific data is organized by the following parameters:

- Transmit L-PDU Handle
- Reference to a data structure, which contains L-PDU related data: L-SDU length (1) and pointer to the L-SDU (2)

The reference to the L-PDU data structure is used as a parameter in several CanIf's API services, e.g. CanIf_Transmit() or the callback service <User_RxIndication>().

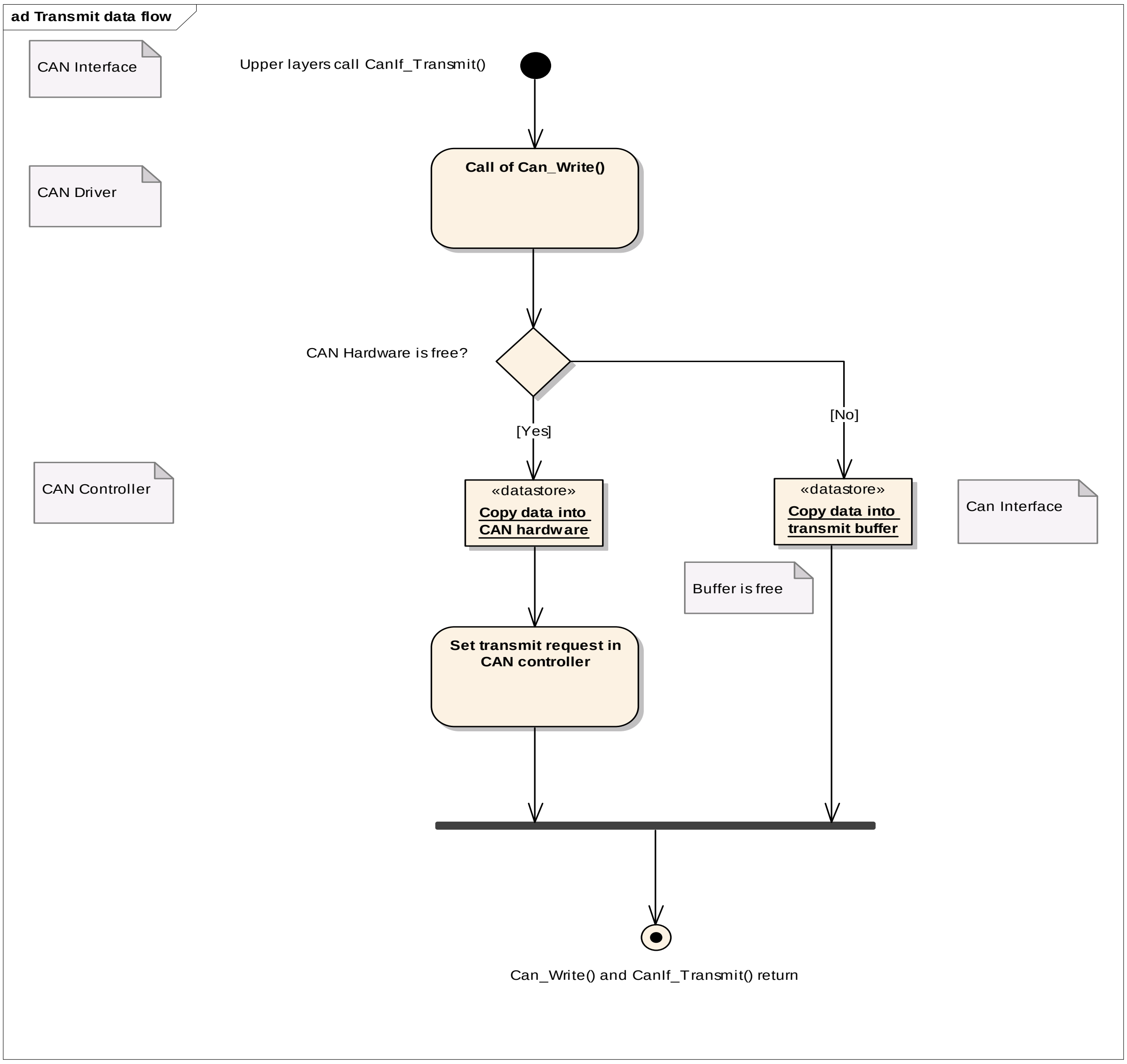

ad Transmit data flow
CAN Interface
CAN Driver
CAN Controller
Upper layers call CanIf_Transmit()
Call of Can_Write()
CAN Hardware is free?
[Yes]
[No]
«datastore»
Copy data into CAN hardware
«datastore»
Copy data into transmit buffer
Can Interface
Buffer is free
Set transmit request in CAN controller
Can_Write() and CanIf_Transmit() return

Figure 8 Transmit data flow

The CanIf stores information about the available hardware objects configured for transmission purposes. The function `CanIf_Transmit()` maps the `CanTxPduId` to the corresponding HTH and calls the function `Can_Write()` (see CANIF318).

7.11 Transmit buffering

7.11.1 General behavior

At the scope of the CanIf the transmit process starts with the call of `CanIf_Transmit()` and it ends with invocation of upper layer module's callback service `<User_TxConfirmation>()`. During the transmit process the CanIf, the CanDrv and the CAN Mailbox altogether shall store the L-PDU to be transmitted only once at a single location. Either in the CAN hardware transmit object or the transmit L-PDU buffer inside the CanIf, if transmit buffering is enabled. A single CanIf Tx L-PDU, requested for transmission, shall never be stored twice. This behavior corresponds to the usual way of periodic communication on the CAN network.

If transmit buffering is enabled, the CanIf will store a CanIf Tx L-PDU in a CanIf transmit L-PDU buffer (CanIfTxBuffer), if it is rejected by the CanDrv at transmission request.

Basically, the overall buffer in CanIf for buffering CanIf Tx L-PDUs consits of one or multiple CanIfTxBuffers (see CANIF832_Conf)). Whereas each CanIfTxBuffer is assigned to one or multiple dedicated HTH's (see CANIF833_Conf) and can be configured to buffer one or multiple CanIf Tx L-PDUs. But as already mentioned above only one instance per CanIf Tx L-PDU can be buffered in the overall amount of CanIfTxBuffers.

The behavior of the CanIf during L-PDU transmission differs whether transmit buffering is enabled in the configuration setup for the corresponding CanIf Tx L-PDU, or not. If transmit buffering is disabled and a transmit request to the CAN Driver module fails (CAN controller mailbox is in use, BasicCAN), the L-PDU is not copied to the CAN controller's mailbox and `CanIf_Transmit()` returns the value `E_NOT_OK`. If transmit buffering is enabled and a transmit request to the CAN Driver module fails, depending on the CanIfTxBuffer configuration the L-PDU can be stored in a CanIfTxBuffer. In this case the API `CanIf_Transmit()` returns the value `E_OK` although the transmission could not be performed. In this case the CanIf takes care of the outstanding transmission of the L-PDU via `CanIf_TxConfirmation()` callback and the upper layer doesn't have to retry the transmit request.

The number of available transmit CanIf Tx L-PDU buffers can be configured completely independent from the number of used transmit L-PDUs defined in the CAN network description file for this ECU.

As per CANIF835 a CanIf Tx L-PDU refers HTHs via the CanIfTxBuffer configuration container (see CANIF832_Conf). This is valid if transmit buffering is not needed as well. In this case, the buffer size (see CANIF834_Conf) of the CanIfTxBuffer has to be set to 0. Then CanIfTxBuffer configuration container is only used to refer a HTH.

7.11.2 Buffer characteristics

CANIF831_Conf, CANIF832_Conf, CANIF833_Conf and CANIF834_Conf describe the possible CanIfTxBuffer configurations.

7.11.2.1 Storage of L-PDUs in the transmit L-PDU buffer

The CanIf tries to store a new transmit L-PDU in the transmit L-PDU buffer only, if
- the CanDrv return `CAN_BUSY` during a call of `Can_Write()` (see CANIF381) or

a pending transmit request was successfully aborted (see CANIF054).

[CANIF063] ⌈The CanIf shall support buffering of a CAN L-PDU handle for BasicCAN transmission in the CanIf, if parameter `CANIF_PUBLIC_TX_BUFFERING` (see CANIF618_Conf) is enabled.⌋(BSW01020)

[CANIF381] ⌈If transmit buffering is enabled (see CANIF063) and if the call of Can_Write()returns with CAN_BUSY, the CanIf shall check if it is possible to buffer the complete CanIf Tx L-PDU,which was requested to be transmitted via Can_Write() in a CanIfTxBuffer.⌋(BSW01126)

When the call of Can_Write()returns with CAN_BUSY, the CanDrv has rejected the requested transmission of the L-PDU (see [8]Specification of CAN Driver) because there is no free HW object available at time of the transmit request (Tx request).

[CANIF835] ⌈When the CanIf checks whether it is possible to buffer a CanIf Tx L-PDU (see CANIF381, CANIF054), this shall only be possible, if the CanIf Tx L-PDU is assigned (see CANIF831_Conf) to a CanIfTxBuffer (see CANIF832_Conf), which is configured with a buffer size (see CANIF834_Conf) bigger than zero.⌋()

The buffer size of any CanIfTxBuffer is only configurable bigger than zero, if transmit buffering is enabled. Additionally the buffer size of a single CanIfTxBuffer is only configurable bigger than zero if the CanIfTxBuffer is not assigned to a FullCAN HTH (see CANIF834_Conf).

[CANIF836] ⌈If it is possible to buffer a CanIf Tx L-PDU, because the buffer size of the assigned CanIfTxBuffer is bigger than zero (see CANIF836), the CanIf shall buffer a CanIf Tx L-PDU in a free buffer element of the assigned CanIfTxBuffer, if the CanIf Tx L-PDU is not already buffered in the CanIfTxBuffer.⌋()

[CANIF068] ⌈If it is possible to buffer a CanIf Tx L-PDU, because the buffer size of the assigned CanIfTxBuffer is bigger than zero (see CANIF836), the CanIf shall overwrite a CanIf Tx L-PDU in the assigned CanIfTxBuffer, if the CanIf Tx L-PDU is already buffered in the CanIfTxBuffer when Can_Write() returns CAN_BUSY.⌋()
CANIF068 implies that a CanIf Tx L-PDU shall not be overwritten in a CanIfTxBuffer in the context of CanIf_CancelTxConfirmation().⌋(BSW01011)

If the order of various transmit requests of different L-PDUs shall be kept, transmit requests of upper layer modules must be connected to previous transmit confirmation notifications. This means that a subsequent L-PDU is requested for transmission by the upper layer modules only, if the transmit confirmation of the previous one was notified by the CanIf.

Note: Additionally the order of transmit requests can differ depending on

- the number of configured hardware transmit objects and
- whether transmit cancellation is supported by the CAN controller or not to avoid inner priority inversion. See [[8] Specification of CAN Driver] for further details.

[CANIF837] ⌈If the buffer size is greater zero, all buffer elements are busy and

`CanIf_Transmit()`is called with a new Pdu (no other instance of the same Pdu is already stored in the buffer), then the new Pdu shall not be stored and

`CanIf_Transmit()`shall return `E_NOT_OK`. ⌋()

7.11.2.2 Clearance of transmit L-PDU buffers

[CANIF386] ⌈The CanIf shall evaluate during transmit confirmation (see (CANIF007), whether pending CanIf Tx L-PDUs are stored within the CanIfTxBuffers, which are assigned to the new free Hardware Transmit Object (see CANIF466). ⌋()

[CANIF668] ⌈If pending CanIf Tx L-PDUs are available in the CanIfTxBuffers as per CANIF386, then the CanIf shall initiate a new transmit request of that pending CanIf Tx L-PDU (of the ones assigned to the new HW Transmit Object) with the highest priority (see CANIF070) by call of `Can_Write()`.⌋()

[CANIF070] ⌈The CAN Interface module shall transmit L-PDUs stored in the transmit L-PDU buffers in priority order (see[18]) per each HTH.⌋()

[CANIF183] ⌈When the CanIf calls the function `Can_Write()` for prioritized L-PDU stored in CanIfTxBuffer and the return value of `Can_Write()` is `E_OK`, then the CanIf shall remove this L-PDU from the transmit L-PDU buffer immediately, before the transmit confirmation returns.⌋()

The behavior specified in CANIF183 simplifies the choice of the new transmit L-PDU stored in the transmit L-PDU buffer.

7.11.2.3 Initialization of transmit L-PDU buffers

[CANIF387] ⌈When function `CanIf_Init()`is called, CanIf shall initialize every transmit L-PDU buffer assigned to the CanIf.⌋()

The requirement CANIF387 is necessary to prevent transmission of old data after restart of the CAN controller.

7.11.3 Data integrity of transmit L-PDU buffers

[CANIF033] ⌈The CanIf shall protect access to transmit L-PDU buffers for all transmit L-PDUs by usage of critical sections.⌋(BSW01114)

In the sequence diagrams in chapter [9 Sequence diagrams], the transmit L-PDU buffer operations, which could be preempted by further transmit L-PDU buffer access operations, are emphasized by messages “ENTER CRITICAL SECTION” and “LEAVE CRITICAL SECTION". This will be realized by entering exclusive areas defined within the BSW Scheduler. These exclusive areas can e.g. configured, that all interrupts will be disabled while the exclusive area is entered. The corresponding services from the BSW Scheduler module are `SchM_Enter_CanIf()` and `SchM_Exit_CanIf()`. The exclusive area, which will be defined within the BSW Scheduler module, will be derived via referencing parameter CANIF_RXPDU_BSWSCH_EXCLAREAID_REF (see CANIF669_Conf) and CANIF_TXPDU_BSWSCH_EXCLAREAID_REF (see CANIF670_Conf).

Rationale: for CANIF033: pre-emptive accesses to the transmit L-PDU buffer cannot always be avoided. Such transmit L-PDU buffer access like storing a new L-PDU or removing transmitted L-PDU may occur preemptively.

7.12 Transmit confirmation

7.12.1 Confirmation after transmission completion

If a previous transmit request is completed successfully, the CanDrv notifies it to the CanIf by the call of `CanIf_TxConfirmation()`(CANIF007).

[CANIF383] ⌈When callback notification `CanIf_TxConfirmation()` is called, the CanIf shall identify the upper layer communication layer (see CANIF414), which is linked to the successfully transmitted L-PDU, and shall notify it about the performed transmission by call of CanIf's transmit confirmation service `<User_TxConfirmation>()` (refer to [7.12Transmit confirmation]).⌋()

The callback service `<User_TxConfirmation>()` is implemented by the notified upper layer module.

An upper communication layer module can be designed or configured in a way, that transmit confirmations can be processed with single or multiple callback services for different L-PDUs or groups of L-PDUs. All that services are called by the CanIf at transmit confirmation of the corresponding L-PDU transmission request. The transmit L-PDU handle enables to dispatch different confirmation services associated to the target upper layer module. This assignment is made statically during configuration.

One transmit L-PDU can only be assigned to one single transmit confirmation callback service. Please refer to chapter [8.6.3.1 <User_TxConfirmation>].

[CANIF740] ⌈ If CANIF_PUBLIC_TXCONFIRM_POLLING_SUPPORT (see CANIF733_Conf) is enabled, the CanIf shall buffer the information about a received TxConfirmation per CAN controller, if the CCMSM of that controller is in state CANIF_CS_STARTED.⌋()

7.12.2 Confirmation of transmit cancellation

Some CAN controllers provide cancellation of the pending transmit requests of L-PDUs inside their hardware transmit objects of the CAN controller. This feature is used to prevent inner priority inversion, which may for example occur if the priority of an L-PDU requested for transmission is higher than the priority of the L-PDU waiting for transmission in the CAN hardware transmit object.

In that case the pending transmit request within a CAN hardware transmit object is cancelled and replaced by the newly requested L-PDU with higher priority. The CanDrv informs the CanIf about a successful transmit cancellation via `CanIf_CancelTxConfirmation()` (see 8.4.3 CanIf_CancelTxConfirmation).

[CANIF054] ⌈When `CanIf_CancelTxConfirmation()` is called, the CanIf shall check if it is possible to buffer the canceled CanIf Tx L-PDU, which is referenced in parameter `PduInfoPtr` of `CanIf_CancelTxConfirmation(),` inside a CanIfTxBuffer .⌋()
For further information about the CanIfTxBuffer see chapter 7.11 “Transmit buffering”.

7.13 Transmit cancellation

The CanIf shall execute transmissions of all pending transmit requests in the transmit L-PDU buffers in priority order (see CANIF070).
The feature to abort pending transmit L-PDUs within the transmit hardware objects is necessary to avoid inner priority inversion of L-PDU transmitted on the CAN network (for more details refer to [8]Specification of CAN Driver). The mechanism of the transmit process differs, whether hardware cancellation is supported by the CAN controller or not.

7.13.1 Transmit cancellation not supported or not used

The CanIf handles pending transmit L-PDUs as described in chapter [7.11Transmit buffering], if transmit cancellation is disabled by configuration.
There might be following consequences:

- Priority Inversion of the PDUs stored in CanIf and the ones within the hardware objects might occur.
- Due to this delays latencies of L-PDUs can not be guaranteed on the CAN network

7.13.2 Transmit cancellation supported and used

The CanIf handles pending transmit L-PDUs as described in chapter [7.11Transmit buffering], if transmit cancellation is enabled by configuration.

After CanIf called `Can_Write()` the CanDrv might confirm successful transmit cancellation to the CanIf via `CanIf_CancelTxConfirmation()` and passes the L-PDU requested for transmission back to the CanIf's transmit L-PDU buffer. See UML diagram in chapter [9.6].

Dependent on the used CAN controller and the traffic on the network the cancellation of a pending transmit L-PDU inside a CAN hardware object can be delayed and thus it may occur asynchronously.

[CANIF176] ⌈The CanIf shall only store an aborted transmit L-PDU in a CanIfTxBuffer, if it does not contain a newer pending transmit L-PDUs with the same L-PDU handle (refer to 7.11.2.1Storage of L-PDUs in the transmit L-PDU buffer).⌋()

Rationale: This way of L-PDU storage ensures to keep the latest data of several pending transmit L-PDUs with the same L-PDU handle inside the CanIf's transmit L-PDU buffers.

Hint: The CanIf needs to protect all critical accesses out of pre-emptive call contexts like processing of pending transmit requests in the transmit confirmation context the transmit request service is called re-entrant.

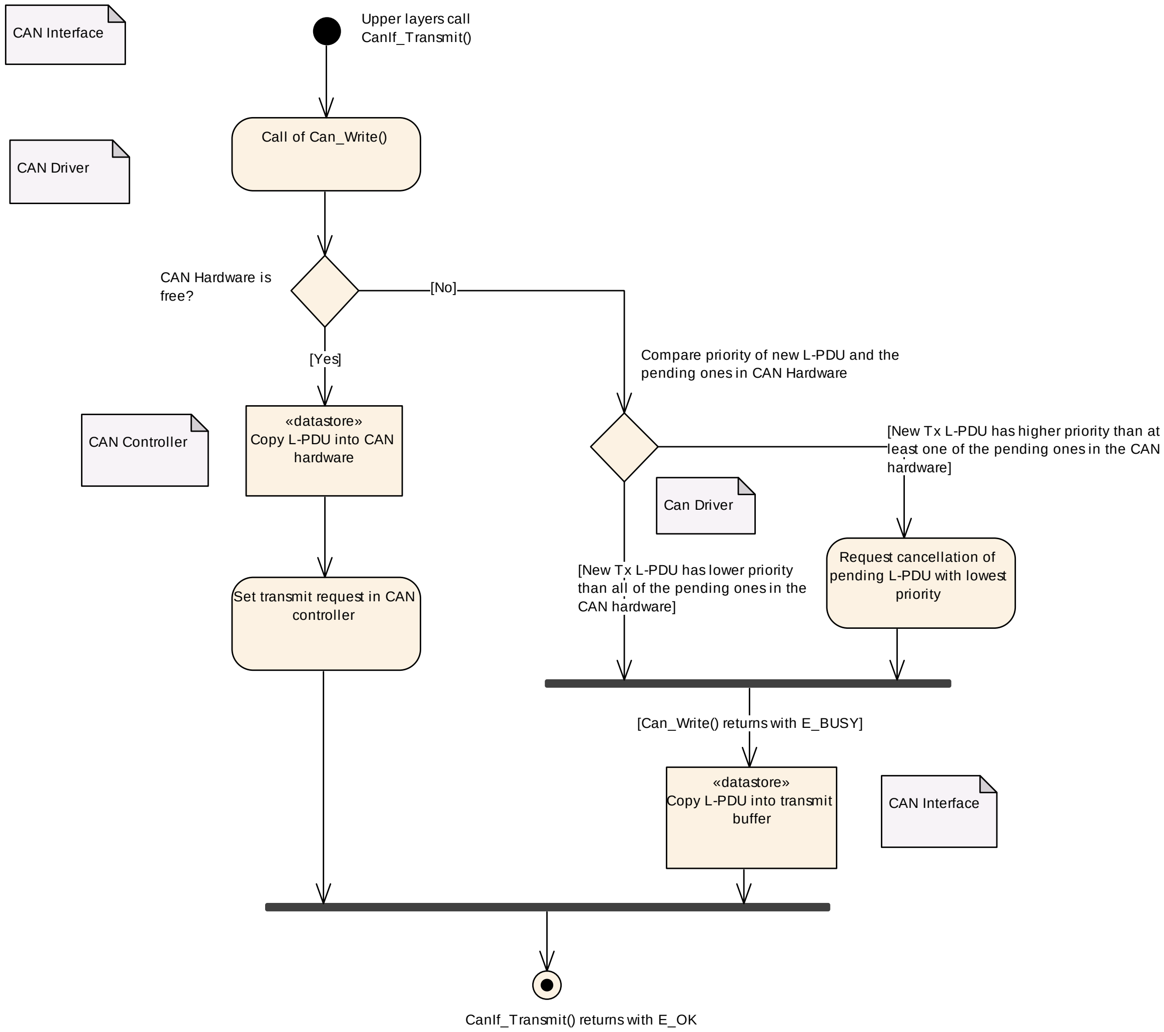

CAN Interface
CAN Driver
Upper layers call CanIf_Transmit()
Call of Can_Write()
CAN Hardware is free?
[No]
[Yes]
CAN Controller
«datastore» Copy L-PDU into CAN hardware
Set transmit request in CAN controller
Compare priority of new L-PDU and the pending ones in CAN Hardware
[New Tx L-PDU has higher priority than at least one of the pending ones in the CAN hardware]
Can Driver
[New Tx L-PDU has lower priority than all of the pending ones in the CAN hardware]
Request cancellation of pending L-PDU with lowest priority
[Can_Write() returns with E_BUSY]
«datastore» Copy L-PDU into transmit buffer
CAN Interface
CanIf_Transmit() returns with E_OK

Figure 9 Transmit cancellation request

In case hardware cancellation is supported and BasicCAN transmission is used inner priority inversion can be avoided and response time predictability can be increased. At FullCAN transmission hardware cancellation is not necessary to avoid inner priority inversion. Please refer to [8]Specification of CAN Driver for more details.

Transmit cancellation can be enabled and disabled by configuration (configuration parameter `CANIF_TX_CANCELLATION`, see CANIF640_Conf). This feature can be activated only, as far as transmit L-PDU buffers have been enabled (configuration parameter `CANIF_PUBLIC_TX_BUFFERING`, see CANIF618_Conf). At configuration time it must be prevented, that transmit cancellation can be enabled, whenever transmit L-PDU buffer configuration is disabled, as specified in field “Dependency” of configuration parameter `CANIF_TX_CANCELLATION` (see CANIF640_Conf).

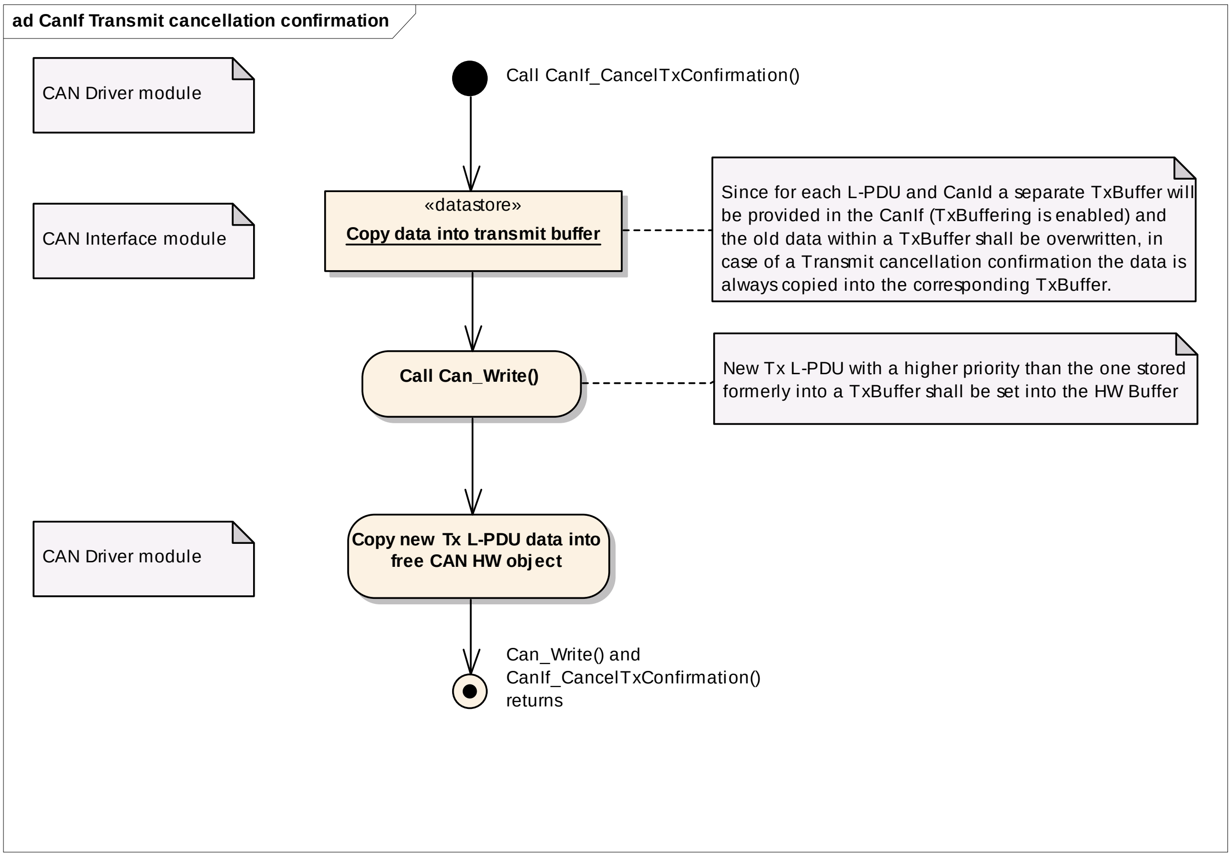

ad CanIf Transmit cancellation confirmation
CAN Driver module
CAN Interface module
CAN Driver module
Call CanIf_CancelTxConfirmation()
«datastore»
Copy data into transmit buffer
Since for each L-PDU and CanId a separate TxBuffer will be provided in the CanIf (TxBuffering is enabled) and the old data within a TxBuffer shall be overwritten, in case of a Transmit cancellation confirmation the data is always copied into the corresponding TxBuffer.
Call Can_Write()
New Tx L-PDU with a higher priority than the one stored formerly into a TxBuffer shall be set into the HW Buffer
Copy new Tx L-PDU data into free CAN HW object
Can_Write() and CanIf_CancelTxConfirmation() returns

Figure 11 Transmit cancellation confirmation

7.14 Receive data flow

7.14.1 Location of PDU data buffers

According to the AUTOSAR Basic Software Architecture the PDU data buffers are placed in the upper layer communication stacks, i.e. AUTOSAR COM, CanNm, CanTp, DCM), where the corresponding data will be evaluated and processed. This means, all transmit as well as all receive PDU buffers are located in these upper layers.

[CANIF057] ⌈The CanIf shall not provide buffers to store SDUs but it shall use the SDU buffers provided by upper layer modules.⌋()

7.14.2 Receive data flow

In case of a new reception of an L-PDU the CanDrv calls `CanIf_RxIndication()` (refer to CANIF006) of the CanIf. The access to the L-PDU specific data is organized by these parameters:

- Hardware Receive Handle (HRH)
- Received CAN Identifier (CanId)
- Received Data Length Code (DLC)
- Reference to the received L-SDU

The received L-SDU is hardware dependent (nibble and byte ordering, access type) and allocated to the lowest layer in the communication system – to the CanDrv.
The HRH serves as a link between the CanDrv and the upper layer module using the L-SDU. The HRH identifies one CAN hardware receive object, where a new CAN L-PDU was received.

After the received L-PDU passed the software filtering (refer to 7.21Software receive filter), identification of the L-PDU handle and passing the DLC Check, the CanIf derives the target upper layer memory buffer location from the L-PDU Handle.
Hereby the hardware receive handle and the L-PDU Handle represents the source and destination information for the copying session of the L-PDU out of the CAN hardware receive object to the L-PDU buffer relocated in the upper layer module.

Initially after detection of a new reception of an L-PDU the CanDrv stores the L-PDU data in an own temporary buffer. If a separate L-SDU normalization is not necessary according to the data structures of the used CAN controller, temporary buffering can be omitted. Thus this feature is up to the CanDrv. The CanIf is not able to recognize, whether the CanDrv uses temporary buffering or a direct hardware access. The CanIf expects normalized L-PDU data in calls of the `CanIf_RxIndication`().

The CAN hardware receive object is locked until the end of the copy process to the temporary or upper layer module buffer. The hardware object will be immediately released after `CanIf_RxIndication`() of the CanIf returns to avoid loss of data.

In case temporary buffering is used, the hardware object remains locked until the data is read out and copied to the temporary buffer. Then the CAN controller is able to perform the next occurred receive event.

In case no temporary buffer is used, the hardware object remains locked until the data is read out and the indication service returns. In this case the parameter of the receive indication callback `CanIf_RxIndication`() refers to the locked CAN RAM with received data.

When `CanIf_RxIndication()` is called, the CanIf identifies the corresponding upper layer module and calls `<User_RxIndication>()` (refer to 8.6.3.2 <User_RxIndication>) of it (see CANIF135).

The temporary buffer or the CAN hardware receive object within the currently received L-PDU remains locked until the end of the copy process. The CanDrv is responsible to unlock them, after CanIf's indication services has returned.

The CanDrv, the CanIf and the upper layer module , which belongs to the received L-PDU, access the same temporary intermediate buffer, which can be located either in the CAN hardware receive object of the CAN controller or as temporary buffer in the CanDrv.

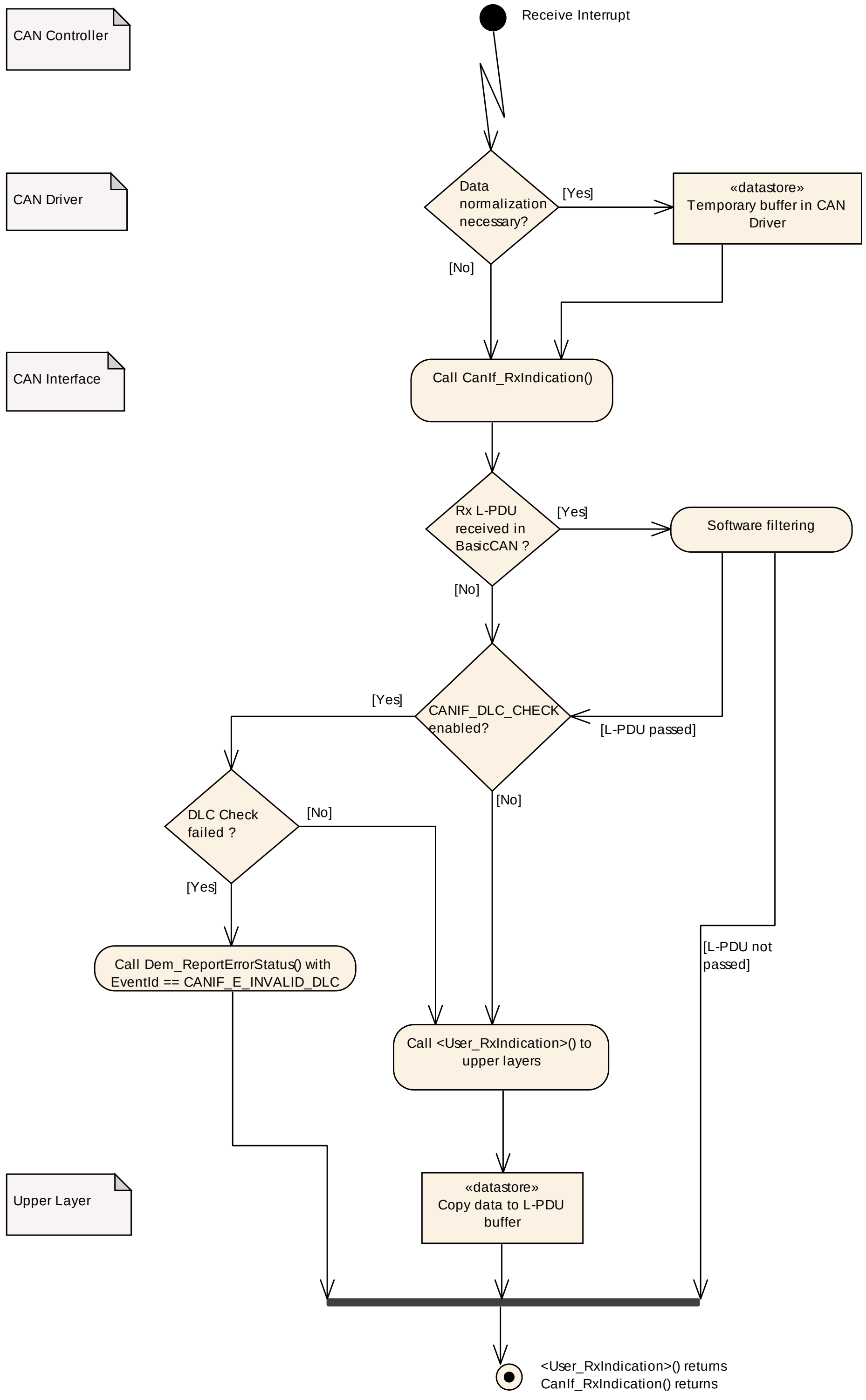

CAN Controller
CAN Driver
CAN Interface
Upper Layer
Receive Interrupt
Data normalization necessary?
[Yes]
[No]
«datastore» Temporary buffer in CAN Driver
Call CanIf_RxIndication()
Rx L-PDU received in BasicCAN ?
[Yes]
[No]
Software filtering
CANIF_DLC_CHECK enabled?
[Yes]
[L-PDU passed]
[No]
DLC Check failed ?
[No]
[Yes]
Call Dem_ReportErrorStatus() with EventId == CANIF_E_INVALID_DLC
[L-PDU not passed]
Call <User_RxIndication>() to upper layers
«datastore» Copy data to L-PDU buffer
<User_RxIndication>() returns
CanIf_RxIndication() returns

Figure 11 Receive data flow

7.15 Receive indication

A call of `CanIf_RxIndication()` (see CANIF006) references in its parameters a newly received CAN L-PDU. If the function `CanIf_RxIndication()` is called, the

CanIf evaluates the CAN L-PDU for acceptance and prepares the CAN L-PDU for later access by the upper communication layers. The CanIf notifies upper layer modules about this asynchronous event using <User_RxIndication>() (see 8.6.3.2 <User_RxIndication>, CANIF012), if configured and if this CAN L-PDU is successfully detected and accepted for further processing. The detailed requirements for this behavior follow here.

[CANIF389] ⌈If the function CanIf_RxIndication() is called, the CanIf shall process the Software Filtering on the received L-PDU as specified in 7.21, if configured (see multiplicity of CANIF628_Conf equals 0..*) If Software Filtering rejects the received L-PDU, the CanIf shall end the receive indication for that call of CanIf_RxIndication().⌋()

[CANIF390] ⌈If the CanIf accepts an L-PDU received via CanIf_RxIndication() during Software Filtering (see CANIF389), the CanIf shall process the DLC check afterwards, if configured (see CANIF617_Conf) .⌋()
For further details, please refer to chapter [7.22 DLC check].

[CANIF297] ⌈ If the CanIf has accepted a L-PDU received via CanIf_RxIndication() during DLC check (see CANIF390), the CanIf shall copy the number of bytes according to the configured DLC value (see CANIF594_Conf, CANIF599_Conf) to the static receive buffer, if configured for that L-PDU (see CANIF198, CANIF600_Conf).⌋()

[CANIF056] ⌈If the CanIf accepts an L-PDU received via CanIf_RxIndication() during DLC check (see CANIF390, CANIF026), the CanIf shall identify if a target upper layer module was configured (see configuration descrption of CANIF012 and CANIF529_Conf, CANIF530_Conf) to be called with its providing receive indication service for the received L-PDU.⌋()

[CANIF135] ⌈If a target upper layer module was configured to be called with its providing receive indication service (see CANIF056), the CanIf shall call this configured receive indication callback service (see CANIF530_Conf) and shall provide the parameters required for upper layer notification callback functions (see CANIF012) based on the parameters of CanIf_RxIndication().⌋(BSW00325)

Note: A single receive L-PDU can only be assigned to a single receive indication callback service (refer to multiplicity of CANIF_USERRXINDICATION_NAME, CANIF530_Conf).

Overview: CanIf performs the following steps at a call of CanIf_RxIndication():

- Software Filtering (only BasicCAN), if configured
- DLC check, if configured
- buffer received L-PDU if configured
- call upper layer receive indication callback service, if configured.

7.16 Read received data

The read received data API CanIf_ReadRxPduData() (see CANIF194) is a common interface for upper layer modules to read CAN L-PDUs recently received from the CAN network. The upper layer modules initiate the receive request only via the CanIf services without direct access to the CanDrv. The initiated receive request is successfully completed, if the CanIf wrote the received CAN L-PDU into the upper layer module L-PDU buffer.

The function CanIf_ReadRxPduData() makes reading out data without dependence of reception event (RxIndication) possible. When it is enabled at configuration time (see CANIF_PUBLIC_READRXPDU_DATA_API,CANIF607_Conf), not necessarily a receive indication service for the same L-PDU has to be configured (see CANIF529_Conf). If needed, the receive indication can be enabled, too.

By this way the type of mechanism to receive CAN L-PDUs (in the upper layer modules of the CanIf) can be chosen at configuration time by the parameter CANIF_RXPDU_USERRXINDICATION_UL (see CANIF529_Conf) and parameter CANIF_RXPDU_READ_DATA (see CANIF600_Conf) according to the needs of the upper layer module, to which the corresponding receive CAN L-PDU belongs to. For details please refer to [9.9 Read received data].

[CANIF198] ⌈ If the configuration parameter CANIF_PUBLIC_READRXPDU_DATA_API (CANIF607_Conf) is set to TRUE, the CanIf shall store each received L-PDU, at which CANIF_RXPDU_READDATA (CANIF600_Conf) is enabled, into a receive L-PDU buffer. This means that if the configuration parameter CANIF_RXPDU_READDATA (CANIF600_Conf) is set to TRUE, the CanIf has to allocate a receive L-PDU buffer for this receive L-PDU.⌋()

[CANIF199] ⌈After call of CanIf_RxIndication() and passing of software filtering and DLC check, the CanIf shall store the received L-PDU in this receive L-PDU buffer. During the call of CanIf_ReadRxPduData() the assigned receive L-PDU buffer containing a recently received L-PDU, the CanIf shall avoid preemptive receive L-PDU buffer access events (refer to CANIF064) to that receive L-PDU buffer. In the sequence diagrams in chapter 9, the receive L-PDU buffer operations, which could be preempted by further receive buffer access operations, are emphasized by messages “ENTER CRITICAL SECTION” and “LEAVE CRITICAL SECTION".⌋()

7.17 Read Tx/Rx notification status

In addition to the notification callback functions the CanIf provides the API service CanIf_ReadTxNotifStatus()(see CANIF202) to read the transmit confirmation status of any transmit CAN L-PDU and the API service

CanIf_ReadRxNotifStatus() is provided to read the receive indication status of any receive CAN L-PDU.

The CanIf's API services CanIf_ReadTxNotifStatus() (see CANIF202) and CanIf_ReadRxNotifStatus() (see CANIF230) can be enabled/disabled globally or per L-PDU at pre-compile time configuration using the configuration parameters
CANIF_PUBLIC_READTXPDU_NOTIFY_STATUS_API (CANIF609_Conf),
CANIF_PUBLIC_READRXPDU_NOTIFY_STATUS_API (CANIF608_Conf),
CANIF_TXPDU_READ_NOTIFYSTATUS (CANIF589_Conf), and
CANIF_RXPDU_READ_NOTIFYSTATUS (CANIF595_Conf).

[CANIF472] ⌈If configuration parameter CANIF_PUBLIC_READTXPDU_NOTIFY_STATUS_API (CANIF609_Conf) is set to TRUE, the CanIf shall store the current notification status for each transmit L-PDU.⌋()

[CANIF473] ⌈If configuration parameter CANIF_PUBLIC_READRXPDU_NOTIFY_STATUS_API (CANIF608_Conf) is set to TRUE, the CanIf shall store the current notification status for each receive L-PDU.⌋()

Rationale for CANIF391 and CANIF393 respectively CANIF392 and CANIF394: This 'read-and-consume' behavior ensures, that at least one successful transmit or receive event occurred after last call of this service.

7.18 Data integrity

[CANIF064] ⌈The CanIf shall protect preemptive events, which access shared resources, that could be changed during the CanIf's event handling, against each other. ⌋(BSW00312)

Rationale: An attempt to update the data in the upper layer module buffers as well as in the internal CanIf's buffers has to be done with respect to possible changes done in the context of an interrupt service routine or other preemptive events. Preemptive events probably occur either from preemptive tasks, multiple CAN interrupts, if multiple physical channels i.e. for gateways are used, or in case of other peripherals or network systems interrupts, which have the needs to transmit and receive CAN L-PDUs on the network.

[CANIF058] ⌈If the CanIf's environment reads data from the CanIf controlled memory areas initiated by calling one of the functions CanIf_Transmit(), CanIf_TxConfirmation(), CanIf_CancelTxConfirmation(), and CanIf_ReadRxPduData(), the CanIf shall guarantee that the provided values are the most recently acquired values.⌋()

Hint: The functions CanIf_Transmit(), CanIf_TxConfirmation(), CanIf_CancelTxConfirmation(), and CanIf_ReadRxPduData() access data from the CanIf controlled memory areas only, if the CanIf is configured to use transmit buffers or receive buffers.

Handling of shared transmit and receive L-PDU buffers are critical issues for the implementation of the CanIf. Therefore the CanIf shall ensure data integrity and thus use appropriate mechanisms for access to shared resources like transmission/reception L-PDU buffers. Preemptive events, i.e. transmission and reception event from other CAN controllers could compromise data integrity by writing into the same L-PDU buffer.

The CanIf can e.g. use the CanDrv services to enable (Can_EnableControllerInterrupts())and disable (Can_DisableControllerInterrupts()) CAN interrupts and its notifications at entry and exit of the critical sections separately for each CAN controller. If there are common resources for multiple CAN controllers, the entire CAN Interrupts must be locked. These sections must not take a long time in order to prevent serious performance degradation. Thus copying of data, change of static variables, counters and semaphores should be carried out inside these critical sections. It is up to the implementation to use appropriate mechanisms to guarantee data integrity, interrupt ability and reentrancy.

The transmit request API CanIf_Transmit() must be able to operate re-entrant to allow multiple transmit request calls caused by different preemptive events of different L-PDU Handles. The CanDrv's transmit request API Can_Write() operates re-entrant as well.

7.19 CAN Controller mode

7.19.1 General functionality

The CanIf provides services for controlling the communication mode of all supported CAN controllers represented by the underlying CanDrv. This means that all CAN controllers are controlled by the corresponding provided API services to request and read the current controller mode.

The CAN controller status information which is stored within the CanIf are accessible via CanIf_GetControllerMode().

The CAN controller status may be changed at request of the upper layer by the calling of CanIf_SetControllerMode() service. The request is validated and passed by the CanIf via the CanDrv API to the addressed CAN controller.

The consistent management of all CAN controllers connected at one CAN network is the task of the CanSm. By this way the CanSm is responsible to set all CAN controllers of one CAN network sequentially to sleep mode or to wake them up.

Hint: Because of CDD, the names of the callback services of the Communication Services are configurable (see chapter 8.6.3). In the following paragraph the usual services of CanSm and EcuM are mentioned.

When a CAN controller signals the network event “BusOff”, the CanIf service `CanIf_ControllerBusOff()` is called which transitions the buffered CAN controller mode (see below CCMSM) in the CanIf to CANIF_CS_STOPPED and which in turn notifies the CanSm by the callback service `CanSm_ControllerBusOff(ControllerId)`.
In case of a CAN bus “wake-up” event the function `CanIf_CheckWakeup(WakeupSource)` may be called during execution of `EcuM_CheckWakeup(WakeupSource)` (see wake-up sequence diagrams of EcuM). The CanIf in turn checks by configured input reference to `EcuMWakeupSource` in the Driver modules, which Driver modules have to be checked. The CanIf gets this information via reference `CanIfCtrlCanCtrlRef` (see CANIF636_Conf).
The Communication Service, which is called, belongs to the service defined during configuration (see CANIF250_Conf). In this way the EcuM as well as the CanSm are able to change CAN controller states and to control the system behavior concerning the BusOff recovery or wakeup procedure.

The state machine in Figure 12 CanIf Controller mode state machine for one CAN controller = CCMSM) gives an overview about the possible CAN controller state transitions, which may be requested by surrounding modules of the CanIf (CanDrv, CanSm, EcuM, CDD etc.). The CanIf does not check these requests for correctness.

The CanIf analyses the function calls `CanIf_ControllerBusOff()` and `CanIf_ControllerModeIndication()` and determines the current mode of the assigned CAN controller, which are represented in the CanIf as states:

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

Requirements describing transitions to one of these CAN Controller mode representing states in detail are structured according to the source state. State `CANIF_CS_INIT` and sub states of `CANIF_CS_STOPPED` are introduced to clarify the different and the common behavior when CAN controller mode changes to `CANIF_CS_STOPPED`, from `CANIF_CS_START` to `CANIF_CS_SLEEP`, or from `CANIF_CS_SLEEP` to `CANIF_CS_START` are requested. Changes of the PDU channel mode are not represented in Figure 12 CanIf Controller mode state machine for one CAN controller).

Figure 13 shows only one sub-state-machine representing the required behavior of one CAN Controller module for sake of lucidity, but there should be a separate sub-state-machine for each assigned CAN Controller module.

The calling modules requesting state transitions of the CCMSM can do this independently of the current state of the CCMSM, i.e. the CanIf accepts every state transition request by calling the function `CanIf_SetControllerMode()`or `CanIf_ControllerBusOff()`. The CanIf does not decide if a requested mode transition of the CAN controller is valid or not. The CanIf only includes the execution of requested mode transitions (see CANIF474).

This network related state machine is implemented in the CanSm. Refer to [11] Specification of CAN State Manager. The CanIf only stores the requested mode and executes the requested transition.

Hint: It has to be regarded that not only the CanSm is able to request CAN controller mode changes.

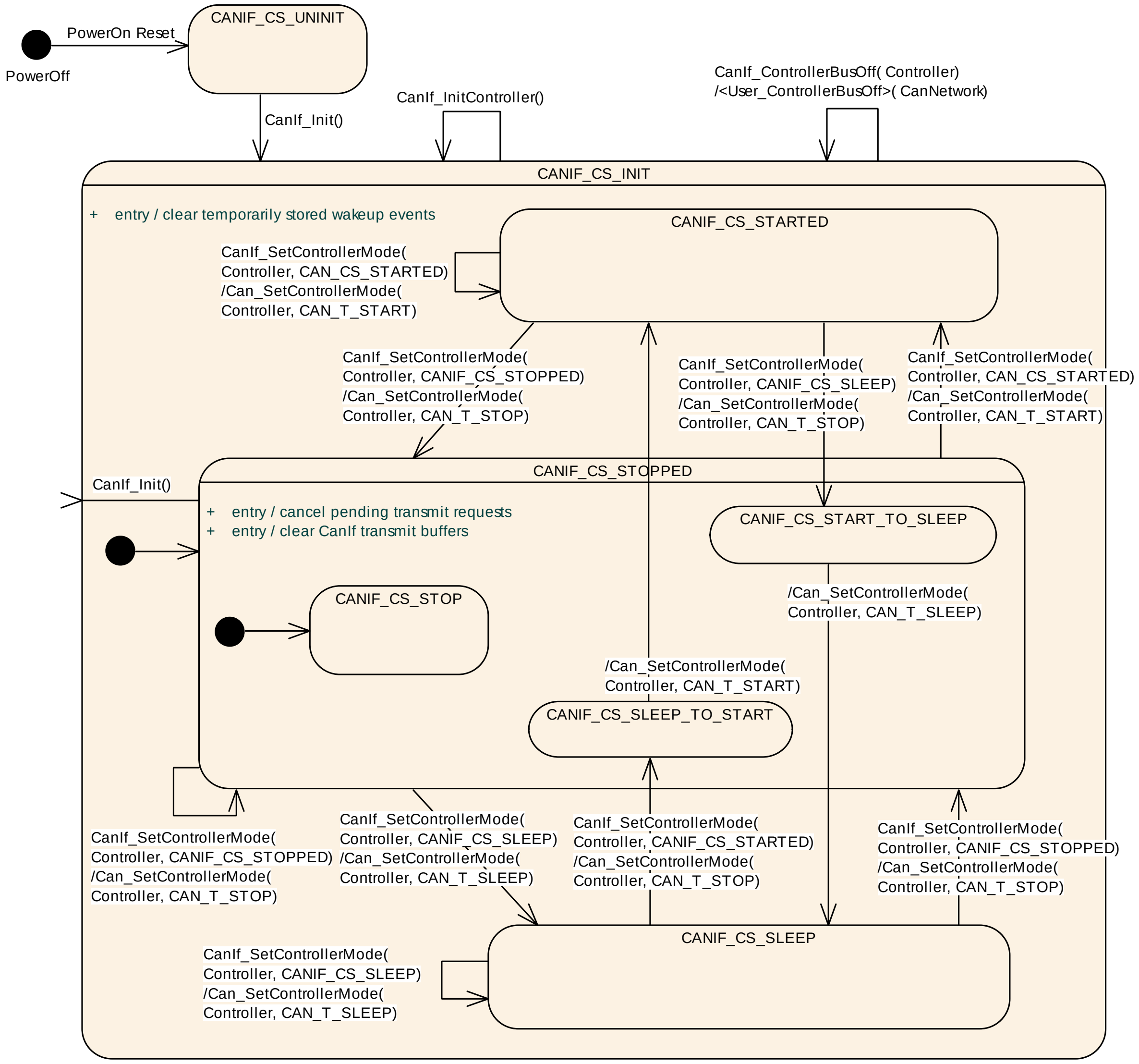


Figure 12 CanIf Controller mode state machine for one CAN controller

General remarks to be considered during implementation:

[CANIF474] ⌈The CAN Interface module shall not contain any complete CAN controller state machine.⌋()

Hint for CANIF474: The CanIf only buffers the modes of the CAN controllers, but it contains no state machine, which checks the transitions.

Because only the CCMSM modes CANIF_CS_UNINIT, CANIF_CS_STOPPED, CANIF_CS_STARTED, and CANIF_CS_SLEEP are visible at the CAN Interface module's interfaces, the additional states of the CCMSM are not mandatory for the implementation of the CanIf.

7.19.2 CAN Controller operation modes

According to the requested operation mode by the CanSm the CanIf translates it into the right order of mode transitions for the CAN controller.
The CanIf changes or stores the new operation mode of the CAN controller after a indication of a successful mode transition via `CanIf_ControllerModeIndication(Controller, ControllerMode)`.

[CANIF475] ⌈If during function `CanIf_SetControllerMode()` the call of `Can_SetControllerMode()`returns with `CAN_NOT_OK`, `CanIf_SetControllerMode()` returns `E_NOT_OK`. ⌋()

7.19.2.1 CANIF_CS_UNINIT

The CanIf is not initialized. The EcuM has to consider, that also the CAN driver module(s) and CAN controller(s) are not initialized.

[CANIF476] ⌈If a CCMSM is in state CANIF_CS_UNINIT when the function `CanIf_Init()` is called, then the CanIf shall take the CCMSM for every assigned CAN controller to state CANIF_CS_INIT.⌋()

7.19.2.2 CANIF_CS_INIT

[CANIF477] ⌈If the CCMSM is in state CANIF_CS_INIT for every assigned CAN controller when the function `CanIf_Init()` is called, then the CAN Interface module shall take the CCMSM for every assigned CAN controller to state CANIF_CS_INIT.⌋()

The explicit transition from CANIF_CS_INIT to CANIF_CS_INIT described in requirement CANIF477 models the reinitialization of the state machine contained within CANIF_CS_INIT.

[CANIF478] ⌈If the state CANIF_CS_INIT of a CCMSM is entered, then the CanIf shall take that CCMSM to sub state CANIF_CS_STOPPED of state CANIF_CS_INIT. ⌋()

[CANIF479] ⌈If a CCMSM enters state CANIF_CS_INIT, then the CanIf shall clear all temporarily stored wakeup events corresponding to that state machine.⌋()

[CANIF298] ⌈ If a CCMSM equals CANIF_CS_INIT when function CanIf_ControllerBusOff(ControllerId) is called with parameter ControllerId referencing that CCMSM, then the CCMSM shall be changed to CANIF_CS_STOPPED.⌋()

7.19.2.2.1 CANIF_CS_STOPPED

The CAN controller cannot receive or transmit CAN L-PDUs on the network in the corresponding mode CAN_T_STOP.

[CANIF480] ⌈If a CCMSM is in state CANIF_CS_STOPPED, when the function CanIf_SetControllerMode(ControllerId, CANIF_CS_STOPPED) is called with parameter ControllerId referencing that CCMSM, then the CanIf shall call Can_SetControllerMode(Controller, CAN_T_STOP). ⌋()

[CANIF713] ⌈If a CCMSM is in state CANIF_CS_STOPPED , when function CanIf_ControllerModeIndication(Controller, ControllerMode) is called with parameter Controller referencing that CCMSM and ControllerMode equals CANIF_CS_STOPPED, then the CanIf shall take the CCMSM to sub state CANIF_CS_STOPPED of state CANIF_CS_INIT.⌋()

[CANIF677] ⌈If a CCMSM is in state CANIF_CS_STOPPED and if the PduIdType parameter in a call of CanIf_Transmit() is assigned to that CAN controller, then the call of CanIf_Transmit()does not result in a call of Can_Write() (see CANIF317) and returns E_NOT_OK (see CANIF005).⌋()

[CANIF481] ⌈If a CCMSM is in state CANIF_CS_STOPPED when the function CanIf_SetControllerMode(ControllerId, CANIF_CS_STARTED) is called with parameter ControllerId referencing that CCMSM, then the CanIf shall call Can_SetControllerMode(Controller, CAN_T_START).⌋()

[CANIF714] ⌈If a CCMSM is in state CANIF_CS_STOPPED , when function CanIf_ControllerModeIndication(Controller, ControllerMode) is called with parameter Controller referencing that CCMSM and ControllerMode equals CANIF_CS_STARTED, then the CanIf shall take the CCMSM to sub state CANIF_CS_STARTED of state CANIF_CS_INIT.⌋()

[CANIF482] ⌈If a CCMSM is in state CANIF_CS_STOPPED when the function CanIf_SetControllerMode(ControllerId, CANIF_CS_SLEEP) is called with parameter ControllerId referencing that CCMSM, then the CAN Interface module shall call Can_SetControllerMode(Controller, CAN_T_SLEEP).⌋()

[CANIF715] ⌈If a CCMSM is in state CANIF_CS_STOPPED , when function CanIf_ControllerModeIndication(Controller, ControllerMode) is called with parameter Controller referencing that CCMSM and ControllerMode equals CANIF_CS_SLEEP, then the CanIf shall take the CCMSM to sub state CANIF_CS_SLEEP of state CANIF_CS_INIT.⌋()

[CANIF485] ⌈If a CCMSM enters state CANIF_CS_STOPPED, then the CanIf shall clear the CanIf transmit buffers assigned to the CAN controller corresponding to that state machine.⌋()

7.19.2.2.2 CANIF_CS_STARTED

In the mode CANIF_CS_STARTED the CanIf passes all transmit requests to the CanDrv and the CanIf can receive CAN L-PDUs and notify upper layers about received L-PDUs.

[CANIF584] ⌈If a CCMSM is in state CANIF_CS_STARTED when the function CanIf_SetControllerMode(ControllerId, CANIF_CS_STARTED) is called with parameter ControllerId referencing that CCMSM, then the CanIf shall call Can_SetControllerMode(Controller, CAN_T_START).⌋()

[CANIF716] ⌈If a CCMSM is in state CANIF_CS_STARTED , when function CanIf_ControllerModeIndication(Controller, ControllerMode) is called with parameter Controller referencing that CCMSM and ControllerMode equals CANIF_CS_STARTED, then the CanIf shall leave the CCMSM in sub state CANIF_CS_STARTED of state CANIF_CS_INIT.⌋()

[CANIF585] ⌈If a CCMSM is in state CANIF_CS_STARTED when the function CanIf_SetControllerMode(ControllerId, CANIF_CS_STOPPED) is called with parameter ControllerId referencing that CCMSM, then the CanIf shall call Can_SetControllerMode(Controller, CAN_T_STOP).⌋()

[CANIF717] ⌈If a CCMSM is in state CANIF_CS_STARTED , when function CanIf_ControllerModeIndication(Controller, ControllerMode) is called with parameter Controller referencing that CCMSM and ControllerMode equals CANIF_CS_STOPPED, then the CanIf shall take the CCMSM to sub state CANIF_CS_STOPPED of state CANIF_CS_INIT.⌋()

[CANIF488] ⌈ If a CCMSM equals CANIF_CS_STARTED when function `CanIf_ControllerBusOff (ControllerId)` is called with parameter `ControllerId` referencing that CCMSM, then the CCMSM shall be changed to CANIF_CS_STOPPED ⌋()

7.19.2.2.3 CANIF_CS_SLEEP

If a CAN controller is set to CAN_T_SLEEP mode, then the controller are enabled, if supported. As long as wake up functionality is not provided by the CAN controller, the CanDrv encapsulates it.

[CANIF486] ⌈ If a CCMSM is in state CANIF_CS_SLEEP when the function `CanIf_SetControllerMode(ControllerId, CANIF_CS_SLEEP)` is called with parameter `ControllerId` referencing that CCMSM, then the CanIf shall call `Can_SetControllerMode(Controller, CAN_T_SLEEP).`⌋()

[CANIF718] ⌈If a CCMSM is in state CANIF_CS_SLEEP , when function `CanIf_ControllerModeIndication(Controller, ControllerMode)` is called with parameter `Controller` referencing that CCMSM and `ControllerMode` equals CANIF_CS_SLEEP, then the CanIf shall leave the CCMSM in sub state CANIF_CS_SLEEP of state CANIF_CS_INIT.⌋()

[CANIF487] ⌈ If a CCMSM is in state CANIF_CS_SLEEP when the function `CanIf_SetControllerMode(ControllerId, CANIF_CS_STOPPED)` is called with parameter `ControllerId` referencing that CCMSM, then the CanIf shall call `Can_SetControllerMode(Controller, CAN_T_WAKEUP).`⌋()

[CANIF719] ⌈If a CCMSM is in state CANIF_CS_SLEEP , when function `CanIf_ControllerModeIndication(Controller, ControllerMode)` is called with parameter `Controller` referencing that CCMSM and `ControllerMode` equals CANIF_CS_STOPPED, then the CanIf shall take the CCMSM to sub state CANIF_CS_STOPPED of state CANIF_CS_INIT.⌋()

When the function `CanIf_SetControllerMode(ControllerId, CANIF_CS_STARTED)` is entered and the CCMSM is in state CANIF_CS_SLEEP, it shall detect an invalid state transition. -> This evaluation has to be made in the CanDrv.

7.19.2.3 BUSOFF

[CANIF739] ⌈ If CANIF_PUBLIC_TXCONFIRM_POLLING_SUPPORT (see CANIF733_Conf) is enabled, the CanIf shall clear the information about a TxConfirmation (see CANIF740), when callback `CanIf_ControllerBusOff(ControllerId)` is called.⌋()

[CANIF724] ⌈When callback `CanIf_ControllerBusOff (ControllerId)` is called, the CanIf shall call `CanSM_ControllerBusOff(ControllerId)` of the CanSm (see chapter 8.6.3.8 or a CDD (see CANIF559, CANIF560)`.` ⌋()

Influence on CCMSM of `CanIf_ControllerBusOff` is described in CANIF298 and CANIF488.

7.19.2.4 Mode Indication

Note: When the callback `CanIf_ControllerModeIndication(Controller, ControllerMode)` is called, the CanIf sets the CCMSM of the corresponding `Controller` to the delivered `ControllerMode` without checking correctness of CCMSM transition.

[CANIF711] ⌈ When callback `CanIf_ControllerModeIndication(Controller, ControllerMode)` is called, the CanIf shall call `CanSm_ControllerModeIndication>(ControllerId, ControllerMode)` of the CanSm (see chapter 8.6.3.8 <User_ControllerModeIndication>) or a CDD (see CANIF691, CANIF692)`.` ⌋()

[CANIF712] ⌈ When callback `CanIf_TrcvModeIndication(Transceiver, TransceiverMode)` is called, the CanIf shall call `CanSM_TransceiverModeIndication(TransceiverId, TransceiverMode)` of the CanSm (see chapter 8.6.3.87 <User_ControllerModeIndication>) or a CDD (see CANIF697, CANIF698)`.` ⌋()

7.19.3 Controller mode transitions

The API for state change requests to the CAN controller behaves in an asynchronous manner with asynchronous notification via callback services.
The real transition to the requested mode occurs asynchronously based on setting of transition requests in the CAN controller hardware, e.g. request for sleep transition `CANIF_CS_SLEEP`. After successful change to e.g. CAN_T_SLEEP mode the CanDrv calls function `CanIf_ControllerModeIndication()`and the CanIf in turn calls function `<User_ControllerModeIndication>()` besides changing the CCMSM to CANIF_CS_SLEEP. If CAN controller transitions very fast,

CanIf_ControllerModeIndication() can be called during CanIf_SetControllerMode(). This is implementation specific.
Unsuccessful or no mode transitions of the CAN controllers have to be tracked by upper layer modules. Mode transitions CANIF_CS_STARTED and CANIF_CS_STOPPED are treated similar.

Upper layer modules of CanIf can poll the current within the CanIf buffered operation mode (CCMSM) by CanIf_GetControllerMode() (see CANIF229).

Not all types of CAN controllers support Sleep and Wake up mode. These modes are then encapsulated by the CanDrv by providing hardware independent operation modes via its interface, which has to be managed by the CanIf.

The CanDrv can release directly a wake up interrupt (to the ECU Integration Code) during the outstanding request Can_SetControllerMode(Controller, CAN_T_SLEEP) and the answer CanIf_ControllerModeIndication(Controller, CANIF_CS_SLEEP), when CAN L-PDUs are transmitted or received at the same time.

This treatment guarantees, that the CanSm is informed immediately about the transition to CANIF_CS_SLEEP mode for handling the CanTrcv and enabling the wake up interrupt.

The CanIf distinguishes between internal initiated CAN controller wake up request (internal request) and network wake up request (external request). The internal request is initiated by call of the CAN Interface module's function CanIf_SetControllerMode(ControllerId, CANIF_CS_STARTED) and it is an internal asynchronous request.
The external request is a CAN controller event, which is notified by the CanDrv or the CanTrcv to the ECU Integration Code. For details see respective UML diagram in the chapter “CAN Wakeup Sequences” of document [15] Specification of ECU State Manager module.

7.19.4 Wake-up

The ECU supports wake-up over CAN network, regardless of the used wake-up method (directly about CAN controller or CAN transceiver), only if the CAN controller and CAN transceiver are set to some kind of “listen for wake-up” mode. This is usually a SLEEP mode, where the usual communication is disabled. Only this mode ensures that the CAN controller is stopped. Thus, the wake-up interrupt can be enabled.

7.19.4.1 Wake-up detection

If wake-up support is enabled (see CANIF180) the CanIf is notified by the Integration Code about a detected CAN wake-up by the service CanIf_CheckWakeup()(see CAN Wakeup Sequences of [15] Specification of ECU State Manager).

[CANIF180] ⌈The CanIf shall provide wake-up service CanIf_CheckWakeup()only, if

- underlying CAN controller provides wake-up support and wake-up is enabled by the parameter CANIF_CONTROLLER_WAKEUP_SUPPORT (see CANIF637_Conf) and by CanDrv configuration.
- underlying CAN transceiver provides wake-up support and wake-up is enabled by the parameter CANIF_TRANSCEIVER_WAKEUP_SUPPORT (see CANIF606_Conf) and CanTrcv configuration.⌋()

[CANIF395] ⌈When CanIf_CheckWakeup(EcuM_WakeupSourceType WakeupSource) is invoked, the CanIf shall querie the CAN controller/transceiver drivers via CanTrcv_CheckWakeup() or Can_CheckWakeup(), which exact CAN hardware device caused the bus wake-up.⌋()

Note: It is implementation specific, which controllers and transceivers are queried. The CanIf just has to find out the exact CAN hardware device.

[CANIF720] ⌈If at least one function call of Can_CheckWakeup() or CanTrcv_CheckWakeup() returns (CAN_OK / E_OK) to the CanIf, then CanIf_CheckWakeup() shall return E_OK.⌋()

[CANIF678] ⌈If all calls of Can_CheckWakeup() or CanTrcv_CheckWakeup() return (CAN_NOT_OK / E_NOT_OK) to the CanIf, then CanIf_CheckWakeup() shall return E_NOT_OK.⌋()

[CANIF679] ⌈If the CCMSM (see chapter 7.19) of the CAN controller, which shall be checked for a wake-up event via CanIf_CheckWakeup(), is not in mode CANIF_CS_SLEEP, the CanIf shall report the development error code CANIF_E_NOT_SLEEP to the Det_ReportError service of the DET module and CanIf_CheckWakeup() shall return E_NOT_OK.⌋()

7.19.4.2 Wake-up validation

Note: When a CAN controller / transceiver detects a bus wake-up event, then this will be notified to the ECU State Manager indirectly. If such a wake-up event needs to be validated, the EcuM (or a CDD) switches on the corresponding CAN controller (CanIf_SetControllerMode()) and transceiver (CanIf_SetTrcvMode()) (For more details see chapter 9 of [15] Specification of ECU State Manager).

Attention: The CanIf notifies the upper layer modules about received messages after the corresponding CCMSM has been transitioned to CANIF_CS_STARTED and the PDU channel mode has been set to CANIF_SET_TX_ONLINE. Thus, it is necessary that the PDU channel mode is not set to CANIF_SET_TX_ONLINE if wake-up validation is required.

Note: As per CAN411 and CAN Controller State Diagram (see [8] Specification of CAN Driver) a direct transition from mode CAN_T_SLEEP to CAN_T_START is not allowed.

[CANIF226] ⌈ The CanIf shall provide wake-up service `CanIf_CheckValidation()`only, if

- underlying CAN controller provides wake-up support and wake-up is enabled by the parameter `CANIF_CTRL_WAKEUP_SUPPORT` (see CANIF637_Conf) and by CanDrv configuration.
- and/orunderlying CAN transceiver provides wake-up support and wake-up is enabled by the parameter `CANIF_TRCV_WAKEUP_SUPPORT` (see CANIF606_Conf) and CanTrcv configuration.
- and configuration parameter `CANIF_PUBLIC_WAKEUP_CHECK_VALIDATION_SUPPORT` (see CANIF611_Conf) is enabled.⌋()

CANIF286: If `CANIF_PUBLIC_WAKEUP_CHECK_VALIDATION_SUPPORT` equals `True` the CanIf enables the detection for CAN wake-up validation. Therefore the CanIf stores the event of the first called `CanIf_RxIndication()` of a CAN controller which has been set to `CANIF_CS_STARTED`.

[CANIF179] ⌈`<User__ValidateWakeupEvent>(sources)` shall be called during `CanIf_CheckValidation(WakeupSource)`, whereas `sources` is set to `WakeupSource`, if the event of the first called `CanIf_RxIndication()` is stored in the CAN Interface module at the corresponding CAN controller.⌋(BSW01136)

Note: The parameter of the function `<User_ValidateWakeupEvent>()` is of type:

- `sources`: `EcuM_WakeupSourceType` (see [15] Specification of ECU State Manager)

[CANIF681] ⌈ If a wake-up event is not validated for the corresponding `WakeupSource` (see CANIF179), then a function call of `CanIf_CheckValidation(WakeupSource)` shall call the function `<User__ValidateWakeupEvent>(sources)`, whereas all bits of `sources` shall be cleared.⌋()

CANIF756: When CC is set to CS_SLEEP the stored event (call of the first `CanIf_RxIndication)` shall be cleared.

7.20 PDU channel mode control

7.20.1 PDU channel groups

Each L-PDU is assigned to one dedicated physical CAN channel connected to one CAN controller and one CAN network. By this way all L-PDUs belonging to one physical channel can be controlled on the view of handling logically single L-PDU channel groups. Those logical groups represent all L-PDUs of one ECU connected to one underlying CAN network.

The figure below shows one possible usage of L-PDU channel group and its relation to the upper layers and/or networks:

An L-PDU can only be assigned to one channel group.

Typical users like PDU Router or the network management are responsible for controlling the PDU operation modes.

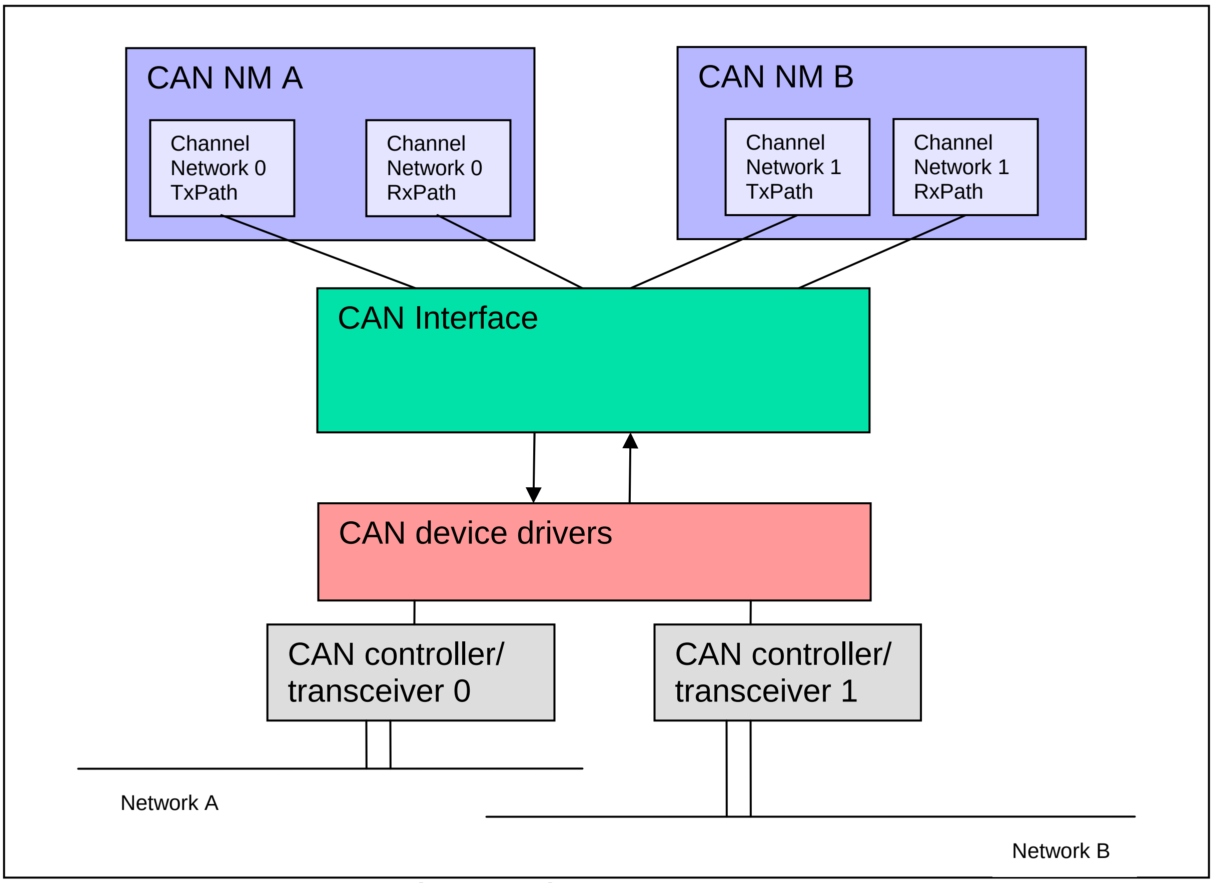


Figure 17 Channel L-PDU groups

7.20.2 PDU channel modes

The CanIf provides the services `CanIf_SetPduMode()` and `CanIf_GetPduMode()` to prevent the processing of

- all transmit L-PDUs of the own ECU belonging to one logical channel,
- all receive L-PDUs of the own ECU belonging to one logical channel,
- all transmit and receive L-PDUs of the own ECU belonging to one logical channel
- all L-PDUs.

Every PDU mode change can be requested for transmission and reception path separately or commonly. A change of the channel mode has only an effect during the network mode CANIF_CS_STARTED (refer to chapter 7.19.2.2.2 CANIF_CS_STARTED]).

The CanIf accepts always requests to change the PDU channel mode independent of its current state. Although this is not necessarily sufficient to e.g. enable transmission of L-PDUs, because the CAN Interface module does not transmit or receive L-PDUs in CANIF_CS_STOPPED, CANIF_CS_SLEEP or CANIF_CS_UNINIT state.

The CANIF_TX_ONLINE/ CANIF_RX_ONLINE PDU channel mode and the CANIF_TX_OFFLINE/ CANIF_RX_OFFLINE PDU channel mode offers the possibility to change the PDU channel mode on the separately for the transmission and reception paths. This modes behave the same like CANIF_SET_ONLINE / CANIF_SET_OFFINE, but only for the transmit L-PDUs or the receive L-PDUs of the corresponding channel.

The CanIf provides information about the status of 'ONLINE'/'OFFLINE' service when required via the service CanIf_GetPduMode().

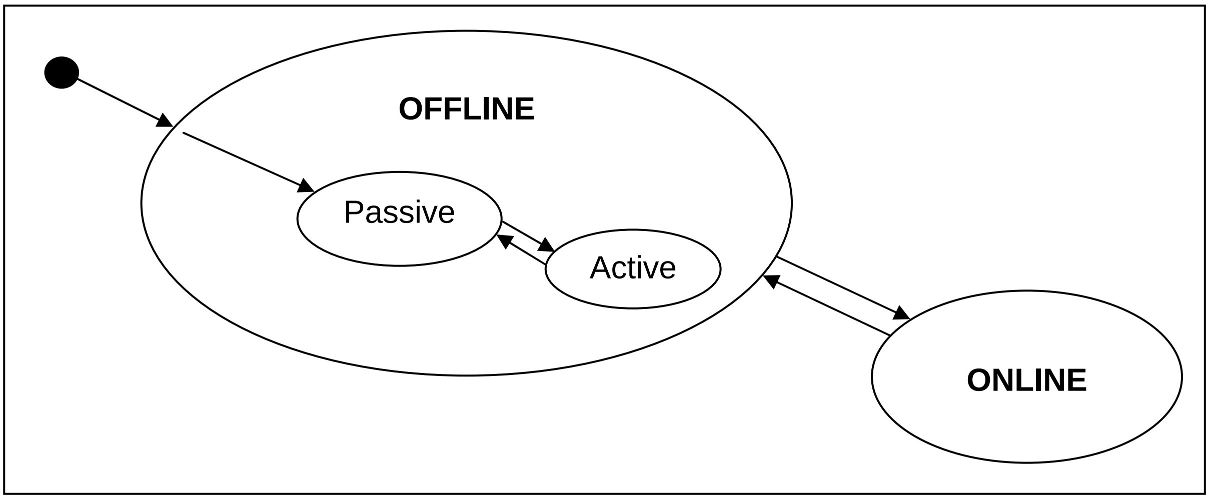


Figure 18 PDU channel mode control

The figure above shows a diagram with possible L-PDU channel modes. Each L-PDU channel can be OFFLINE (no transmission) or ONLINE (activated transmission). A simulation of the successful transmission (transmit confirmation) is supported in the OFFLINE mode and called CANIF_OFFLINE_ACTIVE mode (see CANIF072). The default state of L-PDU channel in OFFLINE mode thus is 'Passive'. No simulation of the successful transmission takes place.

7.20.2.1 CANIF_OFFLINE

[CANIF073] ⌈ After function CanIf_SetPduMode(ControllerId, CANIF_SET_OFFLINE) has been called, the CanIf shall deal with all L-PDUSs, which are assigned to the physical channel (defined by ControllerId ,refer to CANIF382) as follows:

- prevent forwarding of the transmit request calls CanIf_Transmit()to the CanDrv (returning E_NOT_OK to the calling upper layer modules),
- clear the corresponding CanIf transmit buffers,
- prevent invocation of receive indication callback services of the upper layer modules,
- prevent invocation of transmit confirmation callback services of the upper layer modules.⌋()

[CANIF489] ⌈ After function CanIf_SetPduMode(ControllerId, CANIF_SET_TX_OFFLINE) has been called, the CanIf shall deal with the transmit L-PDUSs, which are assigned to the physical channel (defined by ControllerId ,refer to CANIF382) as follows:

- prevent forwarding of the transmit request calls CanIf_Transmit()to the CanDrv (returning E_NOT_OK to the calling upper layer modules),
- clear the corresponding CanIf transmit buffers,
- prevent invocation of transmit confirmation callback services of the upper layer modules.⌋()

[CANIF490] ⌈ After function CanIf_SetPduMode(ControllerId, CANIF_SET_RX_OFFLINE) has been called, the CanIf shall deal with the receive L-PDUSs, which are assigned to the physical channel (defined by ControllerId ,refer to CANIF382) as follows:

- prevent invocation of receive indication callback services of the upper layer modules.⌋()

The BusOff notification is implicitly suppressed in case of CANIF_SET_TX_OFFLINE and CANIF_SET_OFFLINE due to the fact, that in CANIF_SET_TX_OFFLINE and CANIF_SET_OFFLINE mode no L-PDUs can be transmitted and thus the CAN controller is not able to go in BusOff mode by newly requested L-PDUs for transmission.

[CANIF118] ⌈If those transmit L-PDUs, which are already waiting for transmission in the CAN hardware transmit object, will be transmitted immediately after change to CANIF_SET_TX_OFFLINE or CANIF_SET_OFFLINE mode and a subsequent BusOff event occurs, the CanIf does not prohibit execution of the BusOff notification <User_ControllerBusOff>(ControllerId).⌋()

The wake-up notification is not affected concerning mode PDU channel changes.

7.20.2.2 CANIF_ONLINE

[**CANIF075]** ⌈ When function CanIf_SetPduMode(ControllerId, CANIF_SET_ONLINE) has been called, the CanIf shall deal with all L-PDUSs, which are assigned to the physical channel (defined by ControllerId ,refer to CANIF382) as follows:

- enable forwarding of the transmit request calls CanIf_Transmit()to the CanDrv,
- enable invocation of receive indication callback services of the upper layer modules,
- enable invocation of transmit confirmation callback services of the upper layer modules.⌋()

[CANIF491] ⌈ When function CanIf_SetPduMode(ControllerId, CANIF_SET_TX_ONLINE) has been called, the CanIf shall deal with the transmit L-PDUSs, which are assigned to the physical channel (defined by ControllerId ,refer to CANIF382) as follows:

- enable forwarding of the transmit request calls CanIf_Transmit()to the CanDrv,
- enable invocation of transmit confirmation callback services of the upper layer modules.⌋()

[CANIF492] ⌈ When function CanIf_SetPduMode(ControllerId, CANIF_SET_RX_ONLINE) has been called, the CanIf shall deal with the receive L-PDUSs, which are assigned to the physical channel (defined by ControllerId ,refer to CANIF382) as follows:

- enable invocation of receive indication callback services of the upper layer modules.⌋()

7.20.2.3 CANIF_OFFLINE_ACTIVE

The CanIf provides simulation of successful transmission by CANIF_GET_OFFLINE_ACTIVE mode. This mode only affects the transmission path of the CanIf.

The OFFLINE 'Active' mode is enabled by call of CanIf_SetPduMode (ControllerId, CANIF_SET_TX_OFFLINE_ACTIVE). This mode can be left by call of CanIf_SetPduMode(ControllerId, CANIF_SET_ONLINE) or CanIf_SetPduMode(ControlleId, CANIF_SET_TX_OFFLINE).

[CANIF072] ⌈ When function CanIf_SetPduMode(ControllerId, CANIF_SET_TX_OFFLINE_ACTIVE) has been called, the CanIf shall deal with all L-PDUSs, which are assigned to the physical channel (defined by ControllerId ,refer to CANIF382) as follows:

- prevent forwarding of the transmit request calls CanIf_Transmit()to the CanDrv (but not returning E_NOT_OK to the calling upper layer modules),
- enable invocation of transmit confirmation callback services of the upper layer modules synchronously at the end of the transmit request CanIf_Transmit().⌋()

On logical view the CANIF_GET_OFFLINE_ACTIVE mode is a sub-mode of the CANIF_OFFLINE mode, whereas it can be enabled in CANIF_ONLINE as well as in CANIF_OFFLINE mode.

Note: During CANIF_GET_OFFLINE_ACTIVE mode the upper layer has to handle the execution of the transmit confirmations. The transmit confirmation handling is executed immediately at the end of the transmit request (see CANIF072).

Rational: This functionality is useful to realize special operating modes (i.e. diagnosis passive mode) to avoid bus traffic without impact to the notification mechanism. This mode is typically used for diagnostic usage.

7.21 Software receive filter

Not all L-PDUs, which may pass the hardware acceptance filter and therefore are successful received in BasicCAN hardware objects, are defined as receive L-PDUs and thus needed from the corresponding ECU. The CanIf optionally filters out these L-PDUs and prohibits further software processing.

Certain software filter algorithms are provided to optimize software filter runtime. The approach of software filter mechanisms is to find out the corresponding L-PDU handle from the HRH and CAN ID currently being processed. After the L-PDU handle is found, the CanIf accepts the L-PDU and enables upper layers to access L-PDU information directly.

7.21.1 Software filtering concept

The configuration tool handles the information about hardware acceptance filter settings. The most important settings are the number of the L-PDU hardware objects and their range. The outlet range defines, which receive L-PDUs belongs to each hardware receive object. The following definitions are possible:

- a single receive L-PDU (FullCAN reception),
- a list of receive L-PDUs or
- one or multiple ranges of receive L-PDUs can be linked to a hardware receive object (BasicCAN reception).

For definition of range reception it is necessary to define at least one Rx L-PDU with the CanId inside the defined range.

[CANIF645] ⌈A range of CanIds which shall pass the software receive filter shall be defined by its upper limit (see CANIF_HRHRANGE_UPPER_CANID CANIF630_Conf) and lower limit (see CANIF_HRHRANGE_LOWER_CANID CANIF629_Conf) CanId.⌋()

Note: Software receive filtering is optional (see multiplicity of 0..* in CANIF628_Conf).

[CANIF646] ⌈Each configurable range of CAN Ids (see CANIF645), which shall pass the software receive filter, shall be configurable either for StandardCAN IDs or ExtendedCAN IDs via CANIF_HRHRANGE_CANIDTYPE (see CANIF644_Conf).⌋()

Receive L-PDUs are provided as constant structures statically generated from the communication matrix. They are arranged according to the corresponding hardware acceptance filter, so that there is one single list of receive CanIds for every hardware receive object (HRH). The corresponding list can be derived by the HRH, if multiple BasicCAN objects are used. The subsequent filtering is the search through one list of multiple CanIds by comparing them with the new received CanId. In case of a hit the receive L-PDU handle is derived from the found CanId.

[CANIF030] ⌈If the CanIf has found the CanId of the received L-PDU in the list of receive CanIds for the HRH of the received L-PDU, then the CanIf shall accept this L-PDU and the software filtering algorithm shall derive the receive L-PDU handle from the found CanId.⌋(BSW01018)

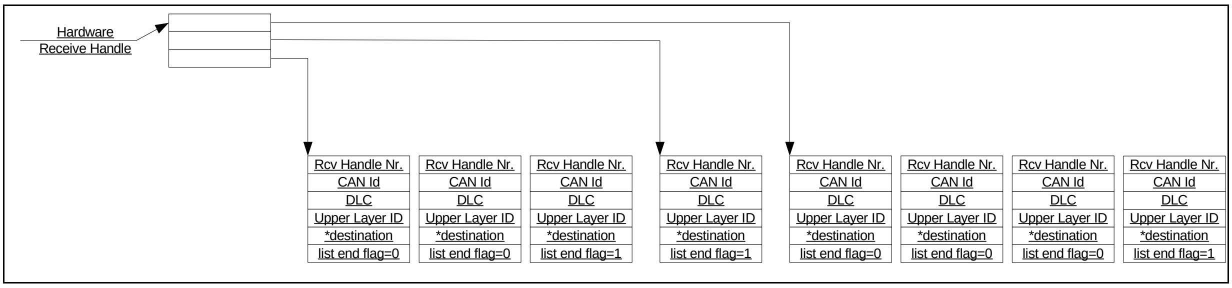


Figure 19 Software filtering example

7.21.2 Software filter algorithms

The choice of suitable software search algorithms it is up to the implementation of the CAN Interface module. According to the wide range of possible receive BasicCAN operations provided by the CAN controller it is recommended to offer several search algorithms like linear search, table search and/or hash search variants to provide the most optimal solution for most use cases.

7.22 DLC check

The received DLC value is compared with the configured DLC value of the received L-PDU. The configured DLC value shall be derived from the size of used bytes inside this L-PDU. The configured DLC value may not be necessarily that DLC value defined in the CAN communication matrix and used by the sender of this CAN L-PDU.

[CANIF026] ⌈The CanIf shall accept all received L-PDUs (see CANIF390) with a DLC value equal or greater then the configured DLC value (see CANIF599_Conf).⌋ (BSW01005)

Hint: The DLC Check can be enabled or disabled globally by CanIf configuration (see parameter CANIF_PRIVATE_DLC_CHECK, CANIF617_Conf) for all used CanDrvs.

[CANIF168] ⌈If the DLC check rejects a received L-PDU (see CANIF026), the CanIf shall report development error code `CANIF_E_INVALID_DLC` to the `Det_ReportError()` service of the DET module.⌋()

[CANIF829] ⌈The CanIf shall pass the received (see CANIF006) length value (DLC) to the target upper layer module (see CANIF135), if the DLC check is passed.⌋()

[CANIF830] ⌈The CanIf shall pass the received (see CANIF006) length value (DLC) to the target upper layer module (see CANIF135), if the DLC check is not configured (see CANIF617_Conf), ⌋()

7.23 L-PDU dispatcher to upper layers

Rationale: At transmission side the L-PDU dispatcher has to find out the corresponding Tx confirmation callback service of the target upper layer module.
At reception side each L-PDU handle belongs to one single upper layer module as destination for the corresponding receive L-PDU or group of such L-PDUs. This relation is assigned statically at configuration time. The task of the L-PDU dispatcher inside of the CanIf is to find out the customer for a received L-PDU and to dispatch the indications towards the found upper layer.
These transmit confirmation as well as receive Indication notification services may exist several times with different names defined in the notified upper layer modules. Those notification services are statically configured, depending on the layers that have to be served.

7.24 Polling mode

The polling mode provides handling of transmit, receive and error events occurred in the CAN hardware without the usage of hardware interrupts. Thus the CanIf and the CanDrv provides notification services for detection and execution corresponding hardware events.
In polling mode the behavior of these CanIf notification services does not change. By this way upper layer modules are abstracted from the strategy to detect hardware events. If different CanDrvs are in use, the calling frequency has to be harmonized during configuration setup and system integration.

These notification services are able to detect new events that occurred in the CAN hardware objects since its last execution. The CanIf's notification services for forwarding of detected events by the CanDrv are the same like for interrupt operation (see chapter 8.4 “Callback notifications”).

The user has to consider, that the CanIf has to be able to perform notification services triggered by interrupt on interrupt level as well as to perform invoked notification services on task level.
If any access to the CAN controller's mailbox is blocked, subsequent transmit buffering takes place (refer [7.12 Transmit buffering]).

The Polling and Interrupt mode can be configured for each underlying CAN controller.

7.25 Multiple CAN Driver support

The CanIf needs a specific mapping to cover multiple CanDrv to provide a common interface to upper layers. Thus, the CanIf must dispatch all actions up-down to the APIs of the corresponding target CanDrv and underlying CAN controller(s) and as well the way down-up by providing multiple callback notifications on the CanIf for multiple CanDrvs.

[CANIF124] ⌈If multiple CanDrvs are assigned to a CanIf, then that CanIf shall provide a separate set of callback function for each CanDrv, in which the callback function names has to follow the naming convention specified in BSW00347.⌋ (BSW00347)

The naming convention is as follows:

```
<CAN Driver module name>_<vendorID>_<Vendor specific API name><driver
abbreviation>()
```

E.g.:

```
Can_99_Ext1
Can_99_Ext2
```

The additional affixes within the function names shall be derived from configuration reference CANIF_DRIVER_NAME_REF (see CANIF638_Conf).

[CANIF224] ⌈If only one CanDrv is assigned to a CanIf, then that CanIf shall provide the set of callback functions for that CanDrv as defined in chapter 8.4.⌋()

The support for multiple CanDrvs can be enabled and disabled by the configuration parameter `CANIF_MULTIPLE_DRIVER_SUPPORT` (see CANIF612_Conf).

7.25.1 Transmit requests by using multiple CAN Drivers

Each transmit L-PDU enables the CanIf to derive the corresponding CAN controller and implicitly the CanDrv serving the affected hardware unit. Resolving of these dependencies is possible because of the construction of the CAN controller handle: it combines CanDrv handle and the corresponding CAN controller in the hardware unit.

At configuration time a mapping table per used CanDrv with references (function pointers) on its API services for the CanIf should be provided. The CanIf needs only to select the corresponding CanDrv in order to call the correct API service. The sequence diagram below demonstrates two transmit requests directed to the different CanDrvs. For an example refer to [7.25.3 Mapping table for multiple CAN Driver handling] below.

A CAN controller handle will be mapped to the CAN controller local logical name (index) and then to the CAN controller handle dedicated to each CAN controller. This mapping is done during configuration phase.

Note: This is only an example. Finally, it is up to the implementation to access the correct APIs of the underlying CanDrvs.

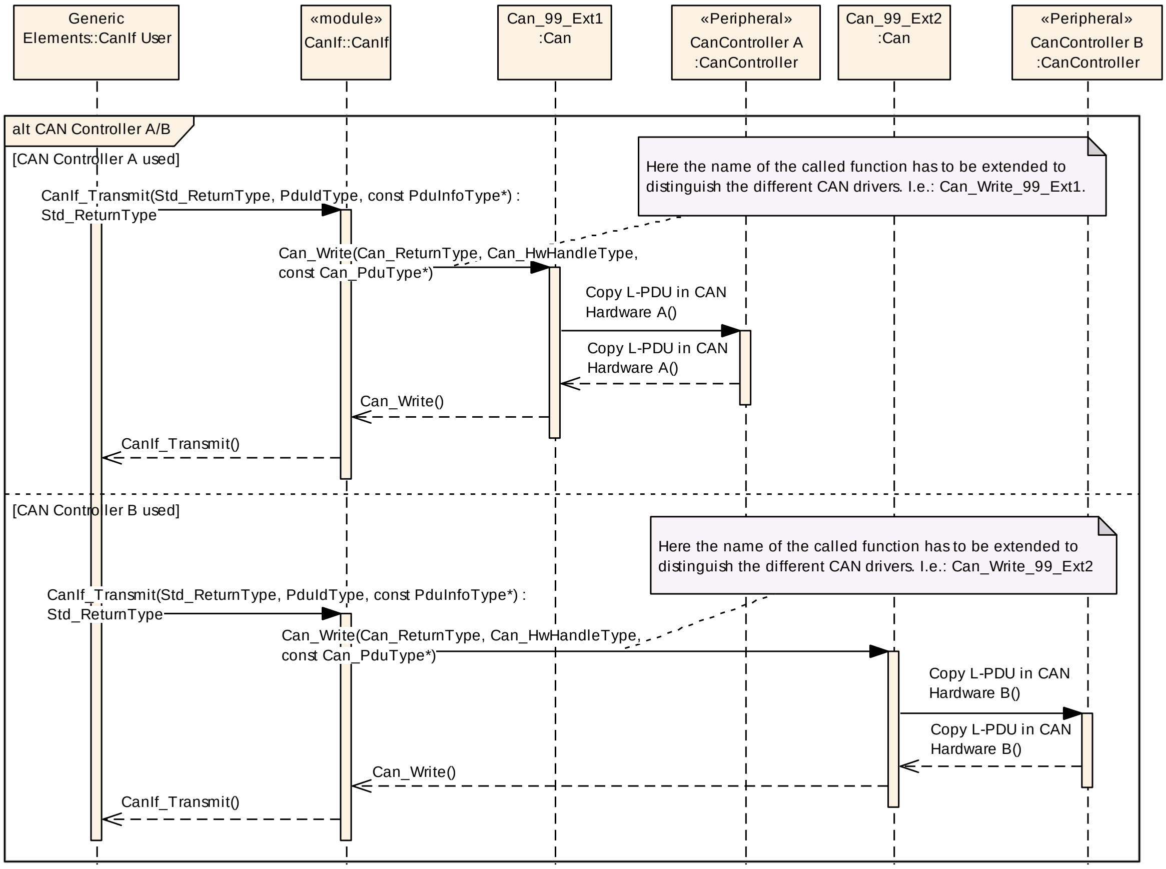

Generic Elements::CanIf User
«module» CanIf::CanIf
Can_99_Ext1 :Can
«Peripheral» CanController A :CanController
Can_99_Ext2 :Can
«Peripheral» CanController B :CanController
alt CAN Controller A/B
[CAN Controller A used]
Here the name of the called function has to be extended to distinguish the different CAN drivers. I.e.: Can_Write_99_Ext1.
CanIf_Transmit(Std_ReturnType, PduIdType, const PduInfoType*) : Std_ReturnType
Can_Write(Can_ReturnType, Can_HwHandleType, const Can_PduType*)
Copy L-PDU in CAN Hardware A()
Copy L-PDU in CAN Hardware A()
Can_Write()
CanIf_Transmit()
[CAN Controller B used]
Here the name of the called function has to be extended to distinguish the different CAN drivers. I.e.: Can_Write_99_Ext2
CanIf_Transmit(Std_ReturnType, PduIdType, const PduInfoType*) : Std_ReturnType
Can_Write(Can_ReturnType, Can_HwHandleType, const Can_PduType*)
Copy L-PDU in CAN Hardware B()
Copy L-PDU in CAN Hardware B()
Can_Write()
CanIf_Transmit()

Figure 16 Transmission request with multiple CAN Drivers - simplified

Operations called	Description

Operations called	Description
CanIf_Transmit (PduId_1, *PduInfoPtr_1)	Upper layer initiates a transmit request. The PduId is used for tracing the requested CAN controller and then to serving the hardware unit. The number of the hardware unit is relevant for the dispatch as it is used as index for the array with pointer to functions. At first the number of the PDU channel group will be extracted from the PduId_1. Each PDU channel group refers to a CAN channel and thus as well the hardware unit number and the CAN controller number. The hardware unit number points on an instance of the CanDrv in the table. This table, created at configuration time, contains all API services configured for the used hardware unit(s). One of these services is the requested transmit service.

Can_Write_99_Ext1
(
`Hth,`
`*PduInfoPtr_1`
)
	Request for transmission to the CAN_Driver_99_Ext1 serving i.e. CAN controller #1 within the "A" hardware unit.

Hardware request	All L-PDU data will be set in Hardware of i.e. CAN controller #0 within hardware unit "A" and the transmit request enabled.
CanIf_Transmit (PduId_2, *PduInfoPtr_2)	Upper layer initiates transmit request. The parameter transmit handle leads to another CAN controller and then to another hardware unit. The number of the hardware unit is relevant for the dispatch as it is used as index for the array with pointer to functions. At first the number of the PDU channel group will be extracted from the PduId_2. Each PDU channel group refers to a CAN channel and thus as well to the hardware unit number and to the CAN controller number. The hardware unit number points on an instance of the CanDrv in the table. This table, created at configuration time, contains all API services configured for the used hardware unit(s). One of these services is the requested transmit service.

Can_Write_99_Ext2
(
`Hth,`
`*PduInfoPtr_2`
)
	Request for transmission to the CAN_Driver_99_Ext2 serving i.e. CAN controller #1 within the "B" hardware unit.

Hardware request	All L-PDU data will be set in the Hardware of i.e. the CAN controller #1 within hardware unit "B" and the transmit request enabled.

7.25.2 Notification mechanism by using multiple CAN Drivers

Every notification callback service invoked by the CanDrvs at the CanIf exists multiple times, if multiple CanDrvs are used in a single ECU. This means, that each used CanDrv calls 'it's own' callback service at the CanIf. The CanIf must provide all callback services unique for each underlying CanDrv. Thus, the HRH parameter is unique at the scope of each CanDrv. Following callback services are affected:

- `CanIf_TxConfirmation`
- `CanIf_RxIndication`
- `CanIf_CancelTxConfirmation`
- `CanIf_ControllerBusOff`
- `CanIf_ControllerModeIndication`

Example: On reception side the corresponding callback routine of the CanDrv are being triggered by the reception events is called at the CanIf. If the CanIf underlies two CanDrvs, the CanIf has to provide two `CanIf_RxIndication()` routines. At configuration time the relation between callback service and used CanDrv has to be set up.

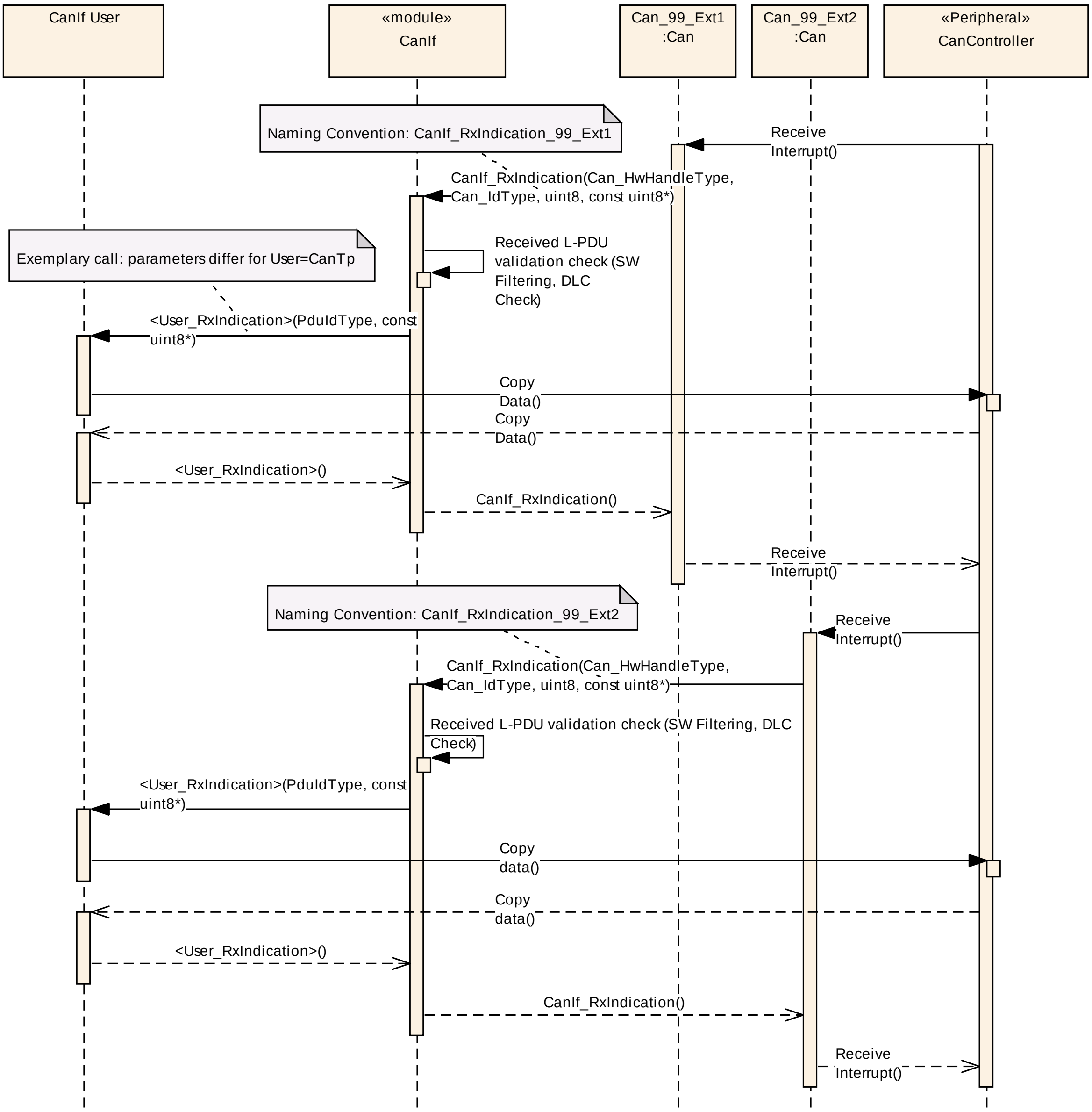

CanIf User
«module»
CanIf
Can_99_Ext1
:Can
Can_99_Ext2
:Can
«Peripheral»
CanController
Naming Convention: CanIf_RxIndication_99_Ext1
Receive Interrupt()
CanIf_RxIndication(Can_HwHandleType, Can_IdType, uint8, const uint8*)
Exemplary call: parameters differ for User=CanTp
Received L-PDU validation check (SW Filtering, DLC Check)
<User_RxIndication>(PduIdType, const uint8*)
Copy Data()
Copy Data()
<User_RxIndication>()
CanIf_RxIndication()
Receive Interrupt()
Naming Convention: CanIf_RxIndication_99_Ext2
Receive Interrupt()
CanIf_RxIndication(Can_HwHandleType, Can_IdType, uint8, const uint8*)
Received L-PDU validation check (SW Filtering, DLC Check)
<User_RxIndication>(PduIdType, const uint8*)
Copy data()
Copy data()
<User_RxIndication>()
CanIf_RxIndication()
Receive Interrupt()

Figure 21 Receive interrupt with multiple CAN Drivers – simplified

Operations called	***Description***
Receive Interrupt	The CAN controller 1 signals a successful reception and triggers a receive interrupt. The ISR of CanDrv A is invoked.
`CanIf_RxIndication_99_Ext1 (Hrh_3, CanId_1, CanDlc_8, *CanSduPtr_1)`	The reception is indicated to the CanIf by calling of `CanIf_RxIndication_99_Ext1()`. The HRH specifies the CAN RAM hardware object and the corresponding CAN controller (`Hrh_3`), which contains the received L-PDU. The temporary buffer is referenced to the CanIf by *CanSduPtr_1.

Operations called	***Description***
Validation check (SW Filtering, DLC Check)	The Software Filtering checks, whether the received L-PDU will be processed on a local ECU. If not, the received L-PDU is not indicated to upper layers. Further processing is suppressed. If the L-PDU is found, the DLC of the received L-PDU is compared with the expected, statically configured one for the received L-PDU.

`<User_RxIndication>`
`(CanRxPduId_4,`
`*CanSduPtr_1)`
	The corresponding receive indication service of the upper layer is called. This signals a successful reception to the target upper layer. The parameter CanRxPduId_4 specifies the L-PDU, the second parameter is the reference on PduInfoType which has the reference on the temporary buffer within the L-SDU.

Receive Interrupt	The CAN controller 2 signals a successful reception and triggers a receive interrupt. The ISR of CanDrv B is invoked.

`CanIf_RxIndication_99_Ext2`
`(Hrh_3, CanId_5,`
`CanDlc_8,`
`*CanSduPtr_2)`
	The reception is indicated to the CanIf by calling of `CanIf_RxIndication_99_Ext2().` The HRH specifies the CAN RAM hardware object and the corresponding CAN controller (`Hrh_3`), which contains the received L-PDU. The temporary buffer is referenced to the CanIf by *CanSduPtr_2.

Validation check (SW Filtering, DLC Check)	The Software Filtering checks, whether the received L-PDU will be processed on a local ECU. If not, the received L-PDU is not indicated to upper layers. Further processing is suppressed. If the L-PDU is found, the DLC of the received L-PDU is compared with the expected, statically configured one for the received L-PDU.

`<User_RxIndication>`
`(CanRxPduId_2,`
`*CanSduPtr_2)`
	The corresponding receive indication service of the upper layer is called. This signals a successful reception to the target upper layer. The parameter CanRxPduId_2 specifies the L-PDU, the second parameter is the reference on PduInfoType which has the reference on the temporary buffer within the L-SDU.

7.25.3 Mapping table for multiple CAN Driver handling

A table with addresses to all CanDrv API services is the basis to provide a unique driver interface to the CanIf. This table makes the assignment from two different driver interfaces to one single driver interface (with prefix (Can_).
In case of L-PDU handle based APIs, the CanIf has to derive the corresponding CanDrv from the L-PDU handle. Afterwards the CanIf can use the CanDrv number as an index for the table with function pointers. The parameters have correspondingly to be translated: i.e. L-PDU handle => HTH/HRH, CanId, Dlc.

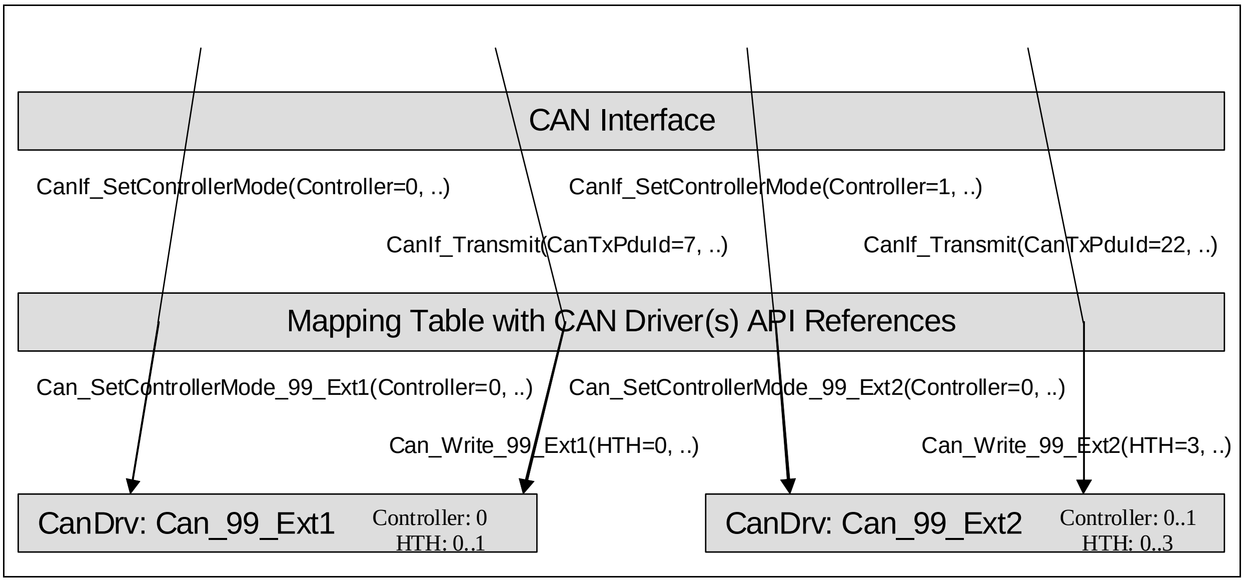

CAN Interface
CanIf_SetControllerMode(Controller=0, ..)
CanIf_SetControllerMode(Controller=1, ..)
CanIf_Transmit(CanTxPduId=7, ..)
CanIf_Transmit(CanTxPduId=22, ..)
Mapping Table with CAN Driver(s) API References
Can_SetControllerMode_99_Ext1(Controller=0, ..)
Can_SetControllerMode_99_Ext2(Controller=0, ..)
Can_Write_99_Ext1(HTH=0, ..)
Can_Write_99_Ext2(HTH=3, ..)
CanDrv: Can_99_Ext1
Controller: 0
HTH: 0..1
CanDrv: Can_99_Ext2
Controller: 0..1
HTH: 0..3

Figure 18 HTH Assignment with multiple CAN Drivers

Each CanDrv supports a certain number of underlying CAN controllers and a fixed number of HTHs. Each CanDrv has an own numbering area, which starts always at zero for controller and HTH.

7.26 Partial Networking

CANIF747: ⌈If Partial Networking (PN) is enabled (see `CANIF_PUBLIC_PN_SUPPORT`, CANIF772_Conf), the CanIf shall support a PnTxFilter per CAN controller which overlays the PDU channel modes.⌋()

CANIF748: ⌈The PnTxFilter of CANIF747 shall only have an effect and transition its modes (enabled/disabled) if more than zero TxPDUs per CAN controller are configured as PnFilterPdu (see `CANIF_TXPDU_PNFILTERPDU`, CANIF773_Conf).⌋()

CANIF749: ⌈If `CanIf_SetPduMode(ControllerId, PduModeRequest)` is called whereas `PduModeRequest` equals `CANIF_SET_ONLINE` or `CANIF_SET_TX_ONLINE` the PnTxFilter of that controller shall be enabled (ref. to CANIF748 and CANIF747).⌋()

CANIF750: ⌈If the PnTxFilter (ref. to CANIF749) of a CAN controller is enabled, the CanIf shall block all Tx requests (return `E_NOT_OK` when `CanIf_Transmit()` is called) to that CAN controller, except if the requested TxPdu is one of the configured PnFilterPdus of that CAN controller. These PnFilterPdus shall always be passed to the corresponding CAN driver module.⌋()

CANIF751: ⌈If `CanIf_TxConfirmation()` is called, the corresponding PnTxFilter shall be disabled (ref. to CANIF748 and CANIF747).⌋()

CANIF752: ⌈If the PnTxFilter of a CAN controller is disabled, the CanIf shall behave as requested via `CanIf_SetPduMode` (see CANIF749).⌋()

Hint (ref. to CANIF752): If e.g. the requested PDU channel mode (see CANIF749) changes in the meantime when PnTxFilter was enabled from `CANIF_SET_ONLINE` to e.g. `CANIF_SET_TX_ONLINE`, the CanIf shall behave correspondingly.

7.27 Error classification

This chapter lists and classifies all errors that can be detected within this software module. Each error is classified according to relevance (development / production) and related error code. For development errors, a value is defined.

[CANIF153] ⌈Values for production code Event Ids are assigned externally by the configuration of the Dem. They are published in the file `Dem_IntErrId.h` and included via `Dem.h.`⌋(BSW00409)

[CANIF154] ⌈Development error values are of type uint8.⌋()

The following table shows the available error codes. The CanIf shall detect them to the DET, if configured.

Type of error	***Relevance***	***Related error code***	***Value***
API service called with invalid parameter	Development	CANIF_E_PARAM_CANID CANIF_E_PARAM_DLC CANIF_E_PARAM_HRH CANIF_E_PARAM_LPDU CANIF_E_PARAM_CONTROLLER CANIF_E_PARAM_CONTROLLERID CANIF_E_PARAM_WAKEUPSOURCE CANIF_E_PARAM_TRCV CANIF_E_PARAM_TRCVMODE CANIF_E_PARAM_TRCVWAKEUPMODE CANIF_E_PARAM_CTRLMODE	10 11 12 13 14 15 16 17 18 19 21
API service called with invalid pointer	Development	CANIF_E_PARAM_POINTER	20
API service used without module initialization	Development	CANIF_E_UNINIT	30
Transmit PDU ID invalid	Development	CANIF_E_INVALID_TXPDUID	50
Receive PDU ID invalid	Development	CANIF_E_INVALID_RXPDUID	60
Failed DLC Check	Development	CANIF_E_INVALID_DLC	61
CAN Interface controller mode state machine is in mode CANIF_CS_STOPPED	Development	CANIF_E_STOPPED	70
CAN Interface controller	Development	CANIF_E_NOT_SLEEP	71

mode state machine is not in mode CANIF_CS_SLEEP			

7.28 Error detection

[CANIF018] ⌈The detection of development errors is configurable (*ON / OFF*) at pre-compile time. The switch `CANIF_PUBLIC_DEV_ERROR_DETECT` (see CANIF614_Conf) shall activate or deactivate the detection of all development errors.⌋ (BSW00369, BSW00386)

[CANIF019] ⌈If the `CANIF_PUBLIC_DEV_ERROR_DETECT` switch is enabled, API checking is enabled. The detailed description of the detected errors can be found in chapter [7.26 Error classification] and chapter [8 API specification]. ⌋(BSW00338, BSW00386, BSW00350)

[CANIF155] ⌈The detection of production code errors cannot be switched off.⌋()

[CANIF661] ⌈If the switch `CANIF_PUBLIC_DEV_ERROR_DETECT` is enabled, all CanIf API services other than `CanIf_Init()` and `CanIf_GetVersion()` shall:

- not execute their normal operation
- report to the DET (using `CANIF_E_UNINIT`)
- and return `E_NOT_OK`

unless the CanIf has been initialized with a preceding call of `CanIf_Init()`.⌋()

7.29 Error notification

[CANIF156] ⌈ Detected development errors shall only be reported to `Det_ReportError` service of the DET, if the pre-processor switch `CANIF_PUBLIC_DEV_ERROR_DETECT` is set to True (see CANIF614_Conf). ⌋ (BSW00386)

Note: If it is mentioned in this document, that `Det_ReportError` service shall be called, this shall only be done if `CANIF_PUBLIC_DEV_ERROR_DETECT` is set to True.

[CANIF020] ⌈Production errors shall be reported to the Dem.⌋(BSW00339)

They shall not be used as the return value of the called function.

[CANIF223] ⌈For all defined production errors it is only required to report the event, when an error or diagnostic relevant event (e.g. state changes, no L-PDU events) occurs. Any status has not to be reported.⌋()

[CANIF119] ⌈Additional errors that are detected because of specific implementation and/or specific hardware properties shall be added in the CanIf specific implementation specification. For doing that, the classification and enumeration listed above can be extended with incremented enumerations.⌋()

7.30 Debugging

[CANIF565] ⌈Each variable that shall be accessible by AUTOSAR Debugging, shall be defined as global variable.⌋()

[CANIF566] ⌈All type definitions of variables which shall be debugged, shall be accessible by the header file CanIf.h.⌋()

[CANIF567] ⌈The declaration of variables in the header file shall be such that it is possible to calculate the size of the variables by C-"sizeof" operation.⌋()

[CANIF568] ⌈Variables available for debugging shall be described in the respective Basic Software Module Description.⌋()

7.31 Published information

[CANIF725] ⌈The standardized common published parameters as required by BSW00402 in the General Requirements on Basic Software Modules [3] shall be published within the header file of this module and need to be provided in the BSW Module Description. The according module abbreviation can be found in the List of Basic Software Modules [1].⌋(BSW00402)

[CANIF726] ⌈The CanIf shall provide a readable module vendor identification in its published parameters (see CANIF725). The naming convention of this module vendor identification for CanIf is `CANIF_VENDOR_ID`. This parameter shall be represented in `uint16` (16 bit).⌋(BSW00374)

[CANIF727] ⌈The CanIf shall provide a module identifier in its published parameters (see CANIF725). The naming convention of this module identifier for CanIf is `CANIF_MODULE_ID`. This parameter shall be represented in `uint16` (16 bit) and it shall be set to the value of CanIf from Basic Software Module list (see [1]).⌋ (BSW00379)

[CANIF728]「The CanIf shall provide the following version numbers with the following naming convention (see CANIF021) in its published parameters (see CANIF725):

- CANIF_SW_MAJOR_VERSION
- CANIF_SW_MINOR_VERSION
- CANIF_SW_PATCH_VERSION
- CANIF_AR_RELEASE_MAJOR_VERSION
- CANIF_AR_RELEASE_MINOR_VERSION
- CANIF_AR_RELEASE_REVISION_VERSION」(BSW00318)

[CANIF729]「The numbering of CANIF_SW_MAJOR_VERSION, CANIF_SW_NINOR_VERSION and CANIF_SW_PATCH_VERSION from CANIF728 shall be vendor specific, but it shall follow requirement BSW00321 from General Requirements on Basic Software Modules [3].」(BSW00321)

Additional module-specific published parameters are listed below if applicable.

8 API specification

8.1 Imported types

In this chapter all types included from the following files are listed.

[CANIF142] ⌈

Module	*Imported Type*
Can	Can_HwHandleType
	Can_IdType
	Can_ReturnType
	Can_StateTransitionType
	Can_PduType
Can_GeneralTypes	CanTrcv_TrcvModeType
	CanTrcv_TrcvWakeupModeType
	CanTrcv_TrcvWakeupReasonType
ComStack_Types	PduIdType
	PduInfoType
EcuM	EcuM_WakeupSourceType
Std_Types	Std_ReturnType
	Std_VersionInfoType

⌋(BSW00348, BSW00353, BSW00361)

8.2 Type definitions

8.2.1 CanIf_ConfigType

Name:	`CanIf_ConfigType`		
Type:	`Structure`		
Element:	`void`	`implementation specific`	The contents of the initialization data structure are CAN interface specific
Description:	This type defines a data structure for the post build parameters of the CAN interface for all underlying CAN drivers. At initialization the CanIf gets a pointer to a structure of this type to get access to its configuration data, which is necessary for initialization.		

[CANIF523] ⌈The initialization data structure for a specific CanIf `CanIf_ConfigType` shall include the definition of canIf public parameters and the definition for each L-PDU handle.⌋()

Note: The definition of CanIf public parameters and the definition for each L-PDU handle are specified in chapter 10.

Note: The definition of CAN Interface public parameters contains:

- Number of transmit L-PDUs
- Number of receive L-PDUs
- Number of dynamic transmit L-PDU handles

Note: The definition for each L-PDU handle contains:

- Handle for transmit L-PDUs

- Handle for receive L-PDUs
- Name of transmit L-PDUs
- Name for receive L-PDUs
- CAN Identifier for static and dynamic transmit L-PDUs
- CAN Identifier for receive L-PDUs
- DLC for transmit L-PDUs
- DLC for receive L-PDUs
- Data buffer for receive L-PDUs in case of polling mode
- Transmit L-PDU handle type

8.2.2 CanIf_ControllerModeType

Name:	`CanIf_ControllerModeType`	
Type:	`Enumeration`	
Range:	`CANIF_CS_UNINIT`	= 0 UNINIT mode. Default mode of the CAN Driver and all CAN controllers connected to one CAN network after power on.
	`CANIF_CS_SLEEP`	SLEEP mode. At least one of all CAN controllers connected to one CAN network are set into the SLEEP mode and can be woken up by request of the CAN Driver or by a network event (must be supported by CAN hardware)
	`CANIF_CS_STARTED`	STARTED mode. All CAN controllers connected to one CAN network are started by the CAN Driver and in full-operational mode.
	`CANIF_CS_STOPPED`	STOPPED mode. At least one of all CAN controllers connected to one CAN network is halted and does not operate on the network.
Description:	Operating modes of the CAN Controller and CAN Driver	

8.2.3 CanIf_PduSetModeType

Name:	`CanIf_PduSetModeType`	
Type:	`Enumeration`	
Range:	`CANIF_SET_OFFLINE`	= 0 Channel shall be set to the offline mode => no transmission and reception
	`CANIF_SET_ONLINE`	Channel shall be set to online mode => full operation mode
	`CANIF_SET_RX_OFFLINE`	Receive path of the corresponding channel shall be disabled
	`CANIF_SET_RX_ONLINE`	Receive path of the corresponding channel shall be enabled
	`CANIF_SET_TX_OFFLINE`	Transmit path of the corresponding channel shall be disabled
	`CANIF_SET_TX_OFFLINE_ACTIVE`	Transmit path of the corresponding channel shall be set to the offline active mode => notifications are processed but transmit requests are blocked.
	`CANIF_SET_TX_ONLINE`	Transmit path of the corresponding channel shall be enabled
Description:	Request for PDU channel group. The request type of the channel defines it's transmit or receive activity. Communication direction (transmission and/or reception) of the channel can be controlled separately or together by upper layers.	

8.2.4 CanIf_PduGetModeType

Name:	CanIf_PduGetModeType	
Type:	Enumeration	
Range:	CANIF_GET_OFFLINE	= 0 Channel is in the offline mode => no transmission and reception
	CANIF_GET_OFFLINE_ACTIVE	Transmit path of the corresponding channel is in the offline active mode => transmit notifications are processed but transmit requests are blocked. The receive path is disabled.
	CANIF_GET_OFFLINE_ACTIVE_RX_ONLINE	Transmit path of the corresponding channel is in the offline active mode => transmit notifications are processed but transmit requests are blocked. The receive path is enabled.
	CANIF_GET_ONLINE	Channel is in the online mode => full operation mode
	CANIF_GET_RX_ONLINE	Receive path of the corresponding channel is enabled and transmit path is disabled.
	CANIF_GET_TX_ONLINE	Transmit path of the corresponding channel is enabled and receive path is disabled.
Description:	Status of the PDU channel group. Current mode of the channel defines its transmit or receive activity. Communication direction (transmission and/or reception) of the channel can be controlled separately or together by upper layers.	

8.2.5 CanIf_NotifStatusType

Name:	CanIf_NotifStatusType	
Type:	Enumeration	
Range:	CANIF_NO_NOTIFICATION	= 0 No transmit or receive event occurred for the requested L-PDU.
	CANIF_TX_RX_NOTIFICATION	The requested Rx/Tx CAN L-PDU was successfully transmitted or received.
Description:	Return value of CAN L-PDU notification status.	

8.3 Function definitions

8.3.1 CanIf_Init

[CANIF001] ⌈

Service name:	CanIf_Init
Syntax:	void CanIf_Init(const CanIf_ConfigType* ConfigPtr)

Service ID[hex]:	0x01	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	ConfigPtr	Pointer to configuration parameter set, used e.g. for post build parameters
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service Initializes internal and external interfaces of the CAN Interface for the further processing.	

⌋(BSW00405, BSW101, BSW00358, BSW00414, BSW01021, BSW01022)

Note: All underlying CAN controllers and transceivers still remain not operational.

Note: The service `CanIf_Init()` is called only by the EcuM.

[CANIF085] ⌈The service `CanIf_Init()` shall initialize the global variables and data structures of the CanIf including flags and buffers.⌋()

Note: If default values of the `CanIf_ConfigType` parameters (8.2.1CanIf_ConfigType) of chapter [10 Configuration specification] are specified, they shall be used for initialization.

[CANIF301] ⌈If a NULL pointer is passed in `ConfigPtr` to the service `CanIf_Init()`, the CanIf shall use the default configuration for the function `CanIf_Init()`.⌋()

Note: In case only one configuration setup is used, a NULL pointer is sufficient to choose the one static existing configuration setup.

[CANIF302] ⌈If parameter `ConfigPtr` of `CanIf_Init()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_POINTER` to the `Det_ReportError` service of the DET module only for post build use cases, when `CanIf_Init()` is called.⌋(BSW00323)

8.3.2 CanIf_SetControllerMode

[CANIF003] ⌈

Service name:	CanIf_SetControllerMode	

Syntax:	`Std_ReturnType CanIf_SetControllerMode(`
	`    uint8 ControllerId,`
	`    CanIf_ControllerModeType ControllerMode`
	`)`	

Service ID[hex]:	0x03	
Sync/Async:	Asynchronous	
Reentrancy:	Reentrant (Not for the same controller)	
Parameters (in):	ControllerId	Abstracted CanIf ControllerId which is assigned to a CAN controller, which is requested for mode transition.

	ControllerMode	Requested mode transition
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Controller mode request has been accepted E_NOT_OK: Controller mode request has not been accepted
Description:	This service calls the corresponding CAN Driver service for changing of the CAN controller mode.	

⌋(BSW01027)

Note: The service `CanIf_SetControllerMode()` initiates a transition to the requested CAN controller mode `ControllerMode` of the CAN controller which is assigned by parameter `ControllerId`.

[CANIF308] ⌈The service `CanIf_SetControllerMode()` shall call `Can_SetControllerMode(Controller, Transition)` for the requested CAN controller. ⌋()

[CANIF311] ⌈If parameter `ControllerId` of `CanIf_SetControllerMode()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_CONTROLLERID` to the `Det_ReportError` service of the DET module, when `CanIf_SetControllerMode()` is called.⌋(BSW00323)

[CANIF774] ⌈If parameter `ControllerMode` of `CanIf_SetControllerMode()` has an invalid value (not `CANIF_CS_STARTED`, `CANIF_CS_SLEEP` or `CANIF_CS_STOPPED`), the CanIfshall report development error code `CANIF_E_PARAM_CTRLMODE` to the `Det_ReportError service` of the DET module, when `CanIf_SetControllerMode()` is called.⌋(BSW00323)

[CANIF312] ⌈Caveats of `CanIf_SetControllerMode()`:

- The CAN Driver module must be initialized after Power ON.
- The CAN Interface module must be initialized after Power ON.⌋()

Note: The ID of the CAN controller is published inside the configuration description of the CanIf.

8.3.3 CanIf_GetControllerMode

[CANIF229] ⌈

Service name:	CanIf_GetControllerMode

Syntax:	`Std_ReturnType CanIf_GetControllerMode(`
	`    uint8 ControllerId,`
	`    CanIf_ControllerModeType* ControllerModePtr`
	`)`

Service ID[hex]:	0x04
Sync/Async:	Synchronous
Reentrancy:	Non Reentrant

Parameters (in):	ControllerId	Abstracted CanIf ControllerId which is assigned to a CAN controller, which is requested for current operation mode.
	ControllerModePtr	Pointer to a memory location, where the current mode of the CAN controller will be stored.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Controller mode request has been accepted. E_NOT_OK: Controller mode request has not been accepted.
Description:	This service reports about the current status of the requested CAN controller.	

⌋(BSW01028)

[CANIF541] ⌈The service `CanIf_GetControllerMode` shall return the mode of the requested CAN controller. This mode is the mode which is buffered within the CAN Interface module (see chapter 7.19.2).⌋()

[CANIF313] ⌈If parameter `ControllerId` of `CanIf_GetControllerMode()` has an invalid, the CanIf shall report development error code `CANIF_E_PARAM_CONTROLLERID` to the `Det_ReportError` service of the DET, when `CanIf_GetControllerMode()` is called.⌋(BSW00323)

[CANIF656] ⌈ If parameter `ControllerModePtr` of `CanIf_GetControllerMode()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_POINTER` to the `Det_ReportError` service of the DET, when `CanIf_GetControllerMode()` is called.⌋(BSW00323)

[CANIF316] ⌈Caveats of `CanIf_GetControllerMode`:

- The CanDrv must be initialized after Power ON.
- The CanIf must be initialized after Power ON.⌋()

Note: The ID of the CAN controller module is published inside the configuration description of the CanIf.

8.3.4 CanIf_Transmit

[CANIF005] ⌈

Service name:	CanIf_Transmit	

Syntax:	`Std_ReturnType CanIf_Transmit(`
	`    PduIdType CanTxPduId,`
	`    const PduInfoType* PduInfoPtr`
	`)`	

Service ID[hex]:	0x05	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	CanTxPduId	L-PDU handle of CAN L-PDU to be transmitted. This handle specifies the corresponding CAN L-PDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device.

	PduInfoPtr	Pointer to a structure with CAN L-PDU related data: DLC and pointer to CAN L-SDU buffer
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Transmit request has been accepted E_NOT_OK: Transmit request has not been accepted
Description:	This service initiates a request for transmission of the CAN L-PDU specified by the CanTxPduId and CAN related data in the L-PDU structure.	

⌋(BSW01008)

Note: The corresponding CAN controller and HTH have to be resolved by the `CanTxPduId`.

[CANIF317] ⌈The service `CanIf_Transmit()` shall not accept a transmit request, if the controller mode is not `CANIF_CS_STARTED` and the channel mode at least for the transmit path is not online or offline active.⌋()

[CANIF318] ⌈The service `CanIf_Transmit()` shall map

- the parameters of the data structure, the L-PDU handle with the identifier `CanTxPduId` refers to (CanID, HTH/HRH of the CAN controller)
- and the pointer `PduInfoPtr` points to (DLC, pointer to CAN L-SDU buffer),

to the corresponding CanDrv and call the function `Can_Write(Hth, *PduInfo).`⌋
()

Note: PduInfoPtr is a pointer to a SDU user memory, CAN Identifier, PDU handle and DLC (see [8] Specification of CAN Driver).

[CANIF243] ⌈The CanIf shall set the 'identifier extension flag' (see [18]ISO11898 – Road vehicles - controller area network (CAN)) of the CanId before the CanIf passes the static predefined CanId to the CanDrv at call of `Can_Write()`. The CanId format type of each CAN L-PDU can be configured by `CANIF_CANIFTXPDUID_CANIDTYPE`, refer to CANIF590_Conf.⌋(BSW01141)

.**[CANIF162]** ⌈If the call of `Can_Write()` returns `E_OK` the transmit request service `CanIf_Transmit()` shall return `E_OK`.⌋()

Note: If the call of `Can_Write()`returns `CAN_NOT_OK`, then the transmit request service `CanIf_Transmit()` shall return `E_NOT_OK`. If the transmit request service `CanIf_Transmit()` returns `E_NOT_OK`, then the upper layer module is responsible to repeat the transmit request.

[CANIF319] ⌈If parameter `CanTxPduId` of `CanIf_Transmit()` has an invalid value,,, the CanIf shall report development error code

CANIF_E_INVALID_TXPDUID to the Det_ReportError service of the DET, when CanIf_Transmit() is called.⌋(BSW00323)

[CANIF320] ⌈If parameter PduInfoPtr of CanIf_Transmit() has an invalid value, the CanIf shall report development error code CANIF_E_PARAM_POINTER to the Det_ReportError service of the DET module, when CanIf_Transmit() is called.⌋(BSW00323)

[CANIF323] ⌈Caveats of CanIf_Transmit():

- During the call of this API the buffer of PduInfoPtr is controlled by the CanIf and this buffer should not be accessed for read/write from another call context. After return of this call the ownership changes to the upper layer.
- The CanIf must be initialized after Power ON.⌋()

8.3.5 CanIf_CancelTransmit

[CANIF520] ⌈

Service name:	CanIf_CancelTransmit	

Syntax:	`Std_ReturnType CanIf_CancelTransmit(`
	`    PduIdType CanTxPduId`
	`)`	

Service ID[hex]:	0x18	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	CanTxPduId	L-PDU handle of CAN L-PDU to be transmitted. This handle specifies the corresponding CAN L-PDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	Always return E_OK
Description:	This is a dummy method introduced for interface compatibility.	

⌋()

Note: The service CanIf_CancelTransmit() has no functionality and is called by the AUTOSAR PduR to achieve bus agnostic behavior.

[CANIF521] ⌈The service CanIf_CancelTransmit() shall be pre-compile time configurable On/Off by the configuration parameter CANIF_PUBLIC_CANCEL_TRANSMIT_SUPPORT.(see CANIF614_Conf) It shall be configured ON if PduRComCancelTransmitSupport is configured as ON.⌋()

[CANIF652] ⌈If parameter CanTxPduId of CanIf_CancelTransmit() has an invalid value, the CanIf shall report development error code

CANIF_E_INVALID_TXPDUID to the Det_ReportError service of the DET, when CanIf_CancelTransmit() is called.⌋(BSW00323)

8.3.6 CanIf_ReadRxPduData

[CANIF194] ⌈

Service name:	CanIf_ReadRxPduData	

Syntax:	`Std_ReturnType CanIf_ReadRxPduData(`
	`    PduIdType CanRxPduId,`
	`    PduInfoType* PduInfoPtr`
	`)`	

Service ID[hex]:	0x06	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	CanRxPduId	Receive L-PDU handle of CAN L-PDU. This handle specifies the corresponding CAN L-PDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device.
Parameters (inout):	None	
Parameters (out):	PduInfoPtr	Pointer to a structure with CAN L-PDU related data: DLC and pointer to CAN L-SDU buffer
Return value:	Std_ReturnType	E_OK: Request for L-PDU data has been accepted E_NOT_OK: No valid data has been received
Description:	This service provides the CAN DLC and the received data of the requested CanRxPduId to the calling upper layer.	

⌋(BSW01125, BSW01129, BSW01129)

[CANIF324] ⌈The function CanIf_ReadRxPduData() shall not accept a request and return E_NOT_OK, if the corresponding CCMSM does not equal CANIF_CS_STARTED and the channel mode is in the receive path online.⌋()

[CANIF325] ⌈If parameter CanRxPduId of CanIf_ReadRxPduData() has an invalid value, e.g. not configured to be stored within CanIf via CANIF_READRXPDU_DATA (CANIF600_Conf), the CanIf shall report development error code CANIF_E_INVALID_RXPDUID to the Det_ReportError service of the DET, when CanIf_ReadRxPduData() is called.⌋(BSW00323)

[CANIF326] ⌈If parameter PduInfoPtr of CanIf_ReadRxPduData() has an invalid value, the CanIf shall report development error code CANIF_E_PARAM_POINTER to the Det_ReportError service of the DET module, when CanIf_ReadRxPduData() is called.⌋(BSW00323)

[CANIF329] ⌈Caveats of CanIf_ReadRxPduData():

- During the call of this API the buffer of PduInfoPtr is controlled by the CanIf and this buffer should not be accessed for read/write from another call context. After return of this call the ownership changes to the upper layer.

- This API must not be used for CanRxPduIds, which are defined to receive multiple CAN-Ids (range reception).
- The CanIf must be initialized after Power ON.⌋()

[CANIF330] ⌈Configuration of CanIf_ReadRxPduData(): This API can be enabled or disabled at pre-compile time configuration by the configuration parameter CANIF_PUBLIC_READRXPDU_DATA_API (CANIF607_Conf).⌋()

8.3.7 CanIf_ReadTxNotifStatus

[CANIF202] ⌈

Service name:	CanIf_ReadTxNotifStatus	
Syntax:	CanIf_NotifStatusType CanIf_ReadTxNotifStatus(PduIdType CanTxPduId)	
Service ID[hex]:	0x07	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	CanTxPduId	L-PDU handle of CAN L-PDU to be transmitted. This handle specifies the corresponding CAN L-PDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	CanIf_NotifStatusType	Current confirmation status of the corresponding CAN Tx L-PDU.
Description:	This service returns the confirmation status (confirmation occured of not) of a specific static or dynamic CAN Tx L-PDU, requested by the CanTxPduId.	

⌋(BSW01130)

Note: This function notifies the upper layer about any transmit confirmation event to the corresponding requested CAN L-PDU.

[CANIF393] ⌈ If configuration parameters CANIF_PUBLIC_READTXPDU_NOTIFY_STATUS_API (CANIF609_Conf) and CANIF_TXPDU_READ_NOTIFYSTATUS (CANIF589_Conf) for the transmitted L-PDU are set to TRUE, and if CanIf_ReadTxNotifStatus() is called, the CanIf shall reset the notification status for the transmitted L-PDU.⌋()

[CANIF331] ⌈If parameter CanTxPduId of CanIf_ReadTxNotifStatus() is out of range or if no status information was configured for this CAN Tx L-PDU, the CanIf shall report development error code CANIF_E_INVALID_TXPDUID to the Det_ReportError service of the DET when CanIf_ReadTxNotifStatus() is called.⌋(BSW00323)

[CANIF334] ⌈Caveats of `CanIf_ReadTxNotifyStatus()`: The CanIf must be initialized after Power ON.⌋()

[CANIF335] ⌈Configuration of `CanIf_ReadTxNotifyStatus()`: This API can be enabled or disabled at pre-compile time configuration globally by the parameter `CANIF_PUBLIC_READTXPDU_NOTIFY_STATUS_API` (see CANIF609_Conf).⌋()

8.3.8 CanIf_ReadRxNotifStatus

[CANIF230] ⌈

Service name:	CanIf_ReadRxNotifStatus	

Syntax:	`CanIf_NotifStatusType CanIf_ReadRxNotifStatus(`
	`    PduIdType CanRxPduId`
	`)`	

Service ID[hex]:	0x08	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	CanRxPduId	L-PDU handle of CAN L-PDU to be received. This handle specifies the corresponding CAN L-PDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	CanIf_NotifStatusType	Current indication status of the corresponding CAN Rx L-PDU.
Description:	This service returns the indication status (indication occurred or not) of a specific CAN Rx L-PDU, requested by the CanRxPduId.	

⌋(BSW01130, BSW01131)

Note: This function notifies the upper layer about any receive indication event to the corresponding requested CAN L-PDU.

[CANIF394] ⌈ If configuration parameters `CANIF_PUBLIC_READRXPDU_NOTIFY_STATUS_API` (CANIF608_Conf) and `CANIF_RXPDU_READ_NOTIFYSTATUS` (CANIF595_Conf) are set to TRUE, and if `CanIf_ReadRxNotifStatus()` is called, then the CAN Interface module shall reset the notification status for the received L-PDU.⌋()

[CANIF336] ⌈If parameter `CanRxPduId` of `CanIf_ReadRxNotifStatus()` is out of range or if Status for `CanRxPduId` was requested whereas `CANIF_READRXPDU_DATA_API` is disabled or if no status information was configured for this CAN Rx L-PDU, the CanIf shall report development error code `CANIF_E_INVALID_RXPDUID` to the `Det_ReportError` service of the DET, when `CanIf_ReadRxNotifStatus()` is called.⌋(BSW00323)

Note: The function `CanIf_ReadRxNotifStatus()` must not be used for `CanRxPduIds`, which are defined to receive multiple CAN-Ids (range reception).

[CANIF339] ⌈Caveats of `CanIf_ReadRxNotifStatus()`:

- The CanIf must be initialized after Power `ON`.⌋()

[CANIF340] ⌈Configuration of `CanIf_ReadRxNotifStatus()`: This API can be enabled or disabled at pre-compile time configuration globally by the parameter `CANIF_PUBLIC_READRXPDU_NOTIFY_STATUS_API` (see CANIF608_Conf).⌋()

8.3.9 CanIf_SetPduMode

[CANIF008] ⌈

Service name:	CanIf_SetPduMode	

Syntax:	`Std_ReturnType CanIf_SetPduMode(`
	`uint8 ControllerId,`
	`CanIf_PduSetModeType PduModeRequest`
	`)`	

Service ID[hex]:	0x09	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	ControllerId	All PDUs of the own ECU connected to the corresponding CanIf ControllerId, which is assigned to a physical CAN controller are addressed.
	PduModeRequest	Requested PDU mode change (see CanIf_PduSetModeType)
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Request for mode transition has been accepted. E_NOT_OK: Request for mode transition has not been accepted.
Description:	This service sets the requested mode at the L-PDUs of a predefined logical PDU channel.	

⌋()

Note: The channel parameter denoting the predefined logical PDU channel can be derived from parameter `ControllerId` of function `CanIf_SetPduMode()`.

[CANIF341] ⌈If parameter `ControllerId` of `CanIf_SetPduMode()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_CONTROLLLERID` to the `Det_ReportError` service of the DET module, when `CanIf_SetPduMode()` is called.⌋(BSW00323)

[CANIF344] ⌈Caveats of `CanIf_SetPduMode()`:

- The CanIf must be initialized after Power `ON`.⌋()

8.3.10 CanIf_GetPduMode

[CANIF009] ⌈

Service name:	CanIf_GetPduMode	

Syntax:	`Std_ReturnType CanIf_GetPduMode(`
	`    uint8 ControllerId,`
	`    CanIf_PduGetModeType* PduModePtr`
	`)`	

Service ID[hex]:	0x0a	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant (Not for the same channel)	
Parameters (in):	ControllerId	All PDUs of the own ECU connected to the corresponding CanIf ControllerId, which is assigned to a physical CAN controller are addressed.
Parameters (inout):	None	
Parameters (out):	PduModePtr	Pointer to a memory location, where the current mode of the logical PDU channel will be stored.
Return value:	Std_ReturnType	E_OK: PDU mode request has been accepted E_NOT_OK: PDU mode request has not been accepted
Description:	This service reports the current mode of a requested PDU channel.	

⌋()

[CANIF346] ⌈If parameter `ControllerId` of `CanIf_GetPduMode()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_CONTROLLERID` to the `Det_ReportError` service of the DET module, when `CanIf_GetPduMode()` is called.⌋(BSW00323)

[CANIF657] ⌈If parameter `PduModePtr` of `CanIf_GetPduMode()` has an invalid value, the CanIfshall report development error code `CANIF_E_PARAM_POINTER` to the `Det_ReportError` service of the DET module, when `CanIf_GetPduMode()` is called.⌋(BSW00323)

[CANIF349] ⌈Caveats of `CanIf_SetPduMode()`:

- The CanIf must be initialized after Power `ON`.⌋()

8.3.11 CanIf_GetVersionInfo

[CANIF158] ⌈

Service name:	CanIf_GetVersionInfo	

Syntax:	`void CanIf_GetVersionInfo(`
	`    Std_VersionInfoType* VersionInfo`
	`)`	

Service ID[hex]:	0x0b	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	VersionInfo	Pointer to where to store the version information of this module.
Return value:	None	

Description:	This service returns the version information of the called CAN Interface module.

⌋(BSW00407, BSW00411)

[CANIF350] ⌈The function `CanIf_GetVersionInfo()` shall return the version information of the called CanIf module. The version information includes:
- Module Id
- Vendor Id
- Vendor specific version numbers (BSW00407).⌋()

Implementation hint: If source code for caller and callee of this function is available this function should be realized as a macro. The macro should be defined in the modules header file.

[CANIF658] ⌈If parameter `VersionInfo` of `CanIf_GetVersionInfo()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_POINTER` to the `Det_ReportError` service of the DET module, when `CanIf_GetVersionInfo()` is called.⌋(BSW00323)

[CANIF351] ⌈Configuration of `CanIf_GetVersionInfo()`: This function shall be pre compile time configurable `On/Off` by the configuration parameter `CANIF_PUBLIC_VERSION_INFO_API` (see CANIF613_Conf).⌋()

8.3.12 CanIf_SetDynamicTxId

[CANIF189] ⌈

Service name:	CanIf_SetDynamicTxId	

Syntax:	`void CanIf_SetDynamicTxId(`
	`    PduIdType CanTxPduId,`
	`    Can_IdType CanId`
	`)`	

Service ID[hex]:	0x0c	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	CanTxPduId	L-PDU handle of CAN L-PDU for transmission. This ID specifies the corresponding CAN L-PDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device.
	CanId	Standard/Extended CAN ID of CAN L-PDU that shall be transmitted.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service reconfigures the corresponding CAN identifier of the requested CAN L-PDU.	

⌋()

[CANIF352] ⌈If parameter `CanTxPduId` of `CanIf_SetDynamicTxId()` has an invalid value, the CanIf shall report development error code

CANIF_E_INVALID_TXPDUID to the Det_ReportError service of the DET module, when CanIf_SetDynamicTxId() is called.⌋(BSW00323)

[CANIF353] ⌈If parameter CanId of CanIf_SetDynamicTxId() has an invalid value, the CanIf shall report development error code CANIF_E_PARAM_CANID to the Det_ReportError service of the DET module, when CanIf_SetDynamicTxId() is called.⌋(BSW00323)

[CANIF355] ⌈ If the CanIf was not initialized before calling CanIf_SetDynamicTxId(), then the function CanIf_SetDynamicTxId() shall not execute a reconfiguration of Tx CanId.⌋()

[CANIF356] ⌈Caveats of CanIf_SetDynamicTxId():

- The CanIf must be initialized after Power ON.
- This function may not be interrupted by CanIf_Transmit(), if the same L-PDU ID is handled.⌋()

[CANIF357] ⌈Configuration of CanIf_SetDynamicTxId(): This function shall be pre compile time configurable On/Off by the configuration parameter CANIF_PUBLIC_SETDYNAMICTXID_API (see CANIF610_Conf).⌋()

8.3.13 CanIf_SetTrcvMode

[CANIF287] ⌈

Service name:	CanIf_SetTrcvMode	

Syntax:	Std_ReturnType CanIf_SetTrcvMode(
	    uint8 TransceiverId,
	    CanTrcv_TrcvModeType TransceiverMode
	)	

Service ID[hex]:	0x0d	
Sync/Async:	Asynchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	TransceiverId	Abstracted CanIf TransceiverId, which is assigned to a CAN transceiver, which is requested for mode transition
	TransceiverMode	Requested mode transition
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Transceiver mode request has been accepted. E_NOT_OK: Transceiver mode request has not been accepted.
Description:	This service changes the operation mode of the tansceiver TransceiverId, via calling the corresponding CAN Transceiver Driver service.	

⌋()

Note: For more details, please refer to the [9] Specification of CAN Transceiver Driver.

[CANIF358] ⌈ The function `CanIf_SetTrcvMode()` shall call the function `CanTrcv_SetOpMode(Transceiver, OpMode)` on the corresponding requested CAN Transceiver Driver module.⌋()

Note: The parameters of the service `CanTrcv_SetOpMode()` are of type:

- `OpMode`: `CanTrcv_TrcvModeType` (desired operation mode)
- `Transceiver` : `uint8` (Transceiver to which function call has to be applied)

(see [9] Specification of CAN Transceiver Driver)

[CANIF538] ⌈If parameter `TransceiverId` of `CanIf_SetTrcvMode()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_TRCV` to the `Det_ReportError` service of the DET, when `CanIf_SetTrcvMode()` is called.⌋(BSW00323)

Note: The mode of a transceiver can only be changed to `CANTRCV_TRCVMODE_STANDBY`, when the former mode of the transceiver has been `CANTRCV_TRCVMODE_NORMAL` (see [9]). But this is not checked by the CanIf.

Note: The mode of a transceiver can only be changed to `CANTRCV_TRCVMODE_SLEEP`, when the former mode of the transceiver has been `CANTRCV_TRCVMODE_STANDBY` (see [9]). But this is not checked by the CanIf.

[CANIF648] ⌈If parameter `TransceiverMode` of `CanIf_SetTrcvMode()` has an invalid value (not `CANTRCV_TRCVMODE_STANDBY`, `CANTRCV_TRCVMODE_SLEEP` or `CANTRCV_TRCVMODE_NORMAL`), the CanIf shall report development error code `CANIF_E_PARAM_TRCVMODE` to the `Det_ReportError` service of the DET module, , when `CanIf_SetTrcvMode()` is called ⌋(BSW00323)

Note: The function `CanIf_SetTrcvMode()` should be applicable to all CAN transceivers with all values of `TransceiverMode` independent, if the transceiver hardware supports these modes or not. This is to ease up the view of the CanIf to the assigned physical CAN channel.

[CANIF362] ⌈Configuration of `CanIf_SetTrcvMode()`: The number of supported transceiver types for each network is set up in the configuration phase (see CanInterfaceTransceiverConfiguration CANIF587_Conf and CanInterfaceTransceiverDriverConfiguration CANIF273_Conf). If no transceiver is used, this function shall not be provided.⌋()

8.3.14 CanIf_GetTrcvMode

[CANIF288] ⌈

Service name:	CanIf_GetTrcvMode
Syntax:	`Std_ReturnType CanIf_GetTrcvMode(`

	`CanTrcv_TrcvModeType* TransceiverModePtr,`
	`uint8 TransceiverId`
	`)`	

Service ID[hex]:	0x0e	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	TransceiverId	Abstracted CanIf TransceiverId, which is assigned to a CAN transceiver, which is requested for current operation mode.
Parameters (inout):	None	
Parameters (out):	TransceiverModePtr	Requested mode of requested network the Transceiver is connected to.
Return value:	Std_ReturnType	E_OK: Transceiver mode request has been accepted. E_NOT_OK: Transceiver mode request has not been accepted.
Description:	This function invokes CanTrcv_GetOpMode and updates the parameter TransceiverModePtr with the value OpMode provided by CanTrcv.	

⌋()

Note: For more details, please refer to the [9] Specification of CAN Transceiver Driver

[CANIF363] ⌈The function `CanIf_GetTrcvMode()` shall call the function `CanTrcv_GetOpMode(Transceiver, OpMode)` on the corresponding requested CAN Transceiver Driver module.⌋()

Note: The parameters of the function CanTrcv_GetOpMode are of type:

- `OpMode`: `CanTrcv_TrcvModeType` (desired operation mode)
- `Transceiver` : `uint8` (Transceiver to which API call has to be applied)

(see [9] Specification of CAN Transceiver Driver)

[CANIF364] ⌈If parameter `TransceiverId` of `CanIf_GetTrcvMode()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_TRCV` to the `Det_ReportError` service of the DET module, when `CanIf_GetTrcvMode()` is called.⌋(BSW00323)

[CANIF650] ⌈If parameter `TransceiverModePtr` of `CanIf_GetTrcvMode()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_POINTER` to the `Det_ReportError` service of the DET module, when `CanIf_GetTrcvMode()` was called.⌋(BSW00323)

[CANIF367] ⌈Configuration of `CanIf_GetTrcvMode()`: The number of supported transceiver types for each network is set up in the configuration phase (see CanInterfaceTransceiverConfiguration CANIF587_Conf and CanInterfaceTransceiverDriverConfiguration CANIF273_Conf). If no transceiver is used, this function shall not be provided.⌋()

8.3.15 CanIf_GetTrcvWakeupReason

[CANIF289] ⌈

Service name:	CanIf_GetTrcvWakeupReason	

Syntax:	`Std_ReturnType CanIf_GetTrcvWakeupReason(`
	`    uint8 TransceiverId,`
	`    CanTrcv_TrcvWakeupReasonType* TrcvWuReasonPtr`
	`)`	

Service ID[hex]:	0x0f	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	TransceiverId	Abstracted CanIf TransceiverId, which is assigned to a CAN transceiver, which is requested for wake up reason.
Parameters (inout):	None	
Parameters (out):	TrcvWuReasonPtr	provided pointer to where the requested transceiver wake up reason shall be returned
Return value:	Std_ReturnType	E_OK: Transceiver wake up reason request has been accepted. E_NOT_OK: Transceiver wake up reason request has not been accepted.
Description:	This service returns the reason for the wake up of the transceiver TransceiverId, via calling the corresponding CAN Transceiver Driver service.	

⌋()

Note: The ability to detect and differentiate the possible wake up reasons depends strongly on the CAN transceiver hardware. For more details, please refer to the [9] Specification of CAN Transceiver Driver.

[CANIF368] ⌈The function `CanIf_GetTrcvWakeupReason()` shall call `CanTrcv_GetBusWuReason(Transceiver, Reason)` on the corresponding requested CanTrcv.⌋()

Note: The parameters of the function `CanTrcv_GetBusWuReason()` are of type:

- `Reason`: `CanTrcv_TrcvWakeupReasonType`
- `Transceiver` : `uint8` (Transceiver to which API call has to be applied)

(see [9] Specification of CAN Transceiver Driver)

[CANIF537] ⌈If parameter `TransceiverId` of `CanIf_GetTrcvWakeupReason()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_TRCV` to the `Det_ReportError` service of the DET module, when `CanIf_GetTrcvWakeupReason()` is called.⌋(BSW00323)

[CANIF649] ⌈If parameter `TrcvWuReasonPtr` of `CanIf_GetTrcvWakeupReason()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_POINTER` to the `Det_ReportError` service of the DET module, when `CanIf_GetTrcvWakeupReason()` is called.⌋(BSW00323)

Note: Please be aware, that if more than one network is available, each network may report a different wake-up reason. E.g. if an ECU uses CAN, a wake-up by CAN may occur and the incoming data may cause an internal wake-up for another CAN network.
The service `CanIf_GetTrcvWakeupReason()` has a “per network” view and does not vote the more important reason or sequence internally. The same may be true if e.g. one transceiver controls the power supply and the other is just powered or un-powered. Then one may be able to return `CANIF_TRCV_WU_POWER_ON`, whereas the other may state e.g. `CANIF_TRCV_WU_RESET`.It is up to the calling module to decide, how to handle the wake-up information.

[CANIF371] ⌈Configuration of `CanIf_GetTrcvWakeupReason()`: The number of supported transceiver types for each network is set up in the configuration phase (see CanInterfaceTransceiverConfiguration CANIF587_Conf and CanInterfaceTransceiverDriverConfiguration CANIF273_Conf). If no transceiver is used, this function shall not be provided.⌋()

8.3.16 CanIf_SetTrcvWakeupMode

[CANIF290] ⌈

Service name:	CanIf_SetTrcvWakeupMode	

Syntax:	`Std_ReturnType CanIf_SetTrcvWakeupMode(`
	`uint8 TransceiverId,`
	`CanTrcv_TrcvWakeupModeType TrcvWakeupMode`
	`)`	

Service ID[hex]:	0x10	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	TransceiverId	Abstracted CanIf TransceiverId, which is assigned to a CAN transceiver, which is requested for wake up notification mode transition.
	TrcvWakeupMode	Requested transceiver wake up notification mode
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Will be returned, if the wake up notifications state has been changed to the requested mode. E_NOT_OK: Will be returned, if the wake up notifications state change has failed or the parameter is out of the allowed range. The previous state has not been changed.
Description:	This function shall call CanTrcv_SetTrcvWakeupMode.	

⌋()

Note: For more details, please refer to [9] Specification of CAN Transceiver Driver.

[CANIF372] ⌈The function `CanIf_SetTrcvWakeupMode()` shall call `CanTrcv_SetWakeupMode (Transceiver, TrcvWakeupMode)` on the corresponding requested CanTrcv.⌋()

Info: The parameters of the function `CanTrcv_SetWakeupMode()` are of type:

- TrcvWakeupMode: CanTrcv_TrcvWakeupModeType (see [9]Specification of CAN Transceiver Driver)
- Transceiver : uint8 (Transceiver to which API call has to be applied)

(see [9] Specification of CAN Transceiver Driver)

Note: The following three paragraphs are already described in the Specification of CanTrcv (see [9]). They describe the behavior of a CanTrcv in the respective transceiver wake-up mode, which is requested in parameter TrcvWakeupMode.

CANIF_TRCV_WU_ENABLE:
If the CanTrcv has a stored wake-up event pending for the addressed CanNetwork, the notification is executed within or immediately after the function CanTrcv_SetTrcvWakeupMode() (depending on the implementation).

CANIF_TRCV_WU_DISABLE:
No notifications for wake-up events for the addressed CanNetwork are passed through the CanTrcv. The transceiver device and the underlying communication driver has to buffer detected wake-up events and raise the event(s), when the wake-up notification is enabled again.

CANIF_TRCV_WU_CLEAR:
If notification of wake-up events is disabled (see description of mode CANIF_TRCV_WU_DISABLE), detected wake-up events are buffered. Calling CanIf_SetTrcvWakeupMode() with parameter CANIF_TRCV_WU_CLEAR clears these bufferd events. Clearing of wake-up events has to be used, when the wake-up notification is disabled to clear all stored wake-up events under control of the higher layers of the CanTrcv.

[CANIF535] ⌈If parameter TransceiverId of CanIf_SetTrcvWakeupMode() has an invalid value, the CanIf shall report development error code CANIF_E_PARAM_TRCV to the Det_ReportError service of the DET module, when CanIf_SetTrcvWakeupMode() is called.⌋(BSW00323)

[CANIF536] ⌈If parameter TrcvWakeupMode of CanIf_SetTrcvWakeupMode() has an invalid value, the CanIf shall report development error code CANIF_E_PARAM_TRCVWAKEUPMODE to the Det_ReportError service of the DET module, when CanIf_SetTrcvWakeupMode() is called.⌋(BSW00323)

[CANIF373] ⌈Configuration of CanIf_SetTrcvWakeupMode(): The number of supported transceiver types for each network is set up in the configuration phase (see CanInterfaceTransceiverConfiguration CANIF587_Conf and CanInterfaceTransceiverDriverConfiguration CANIF273_Conf). If no transceiver is used, this function shall not be provided.⌋()

8.3.17 CanIf_CheckWakeup

[CANIF219] ⌈

Service name:	CanIf_CheckWakeup	

Syntax:	`Std_ReturnType CanIf_CheckWakeup(`
	`    EcuM_WakeupSourceType WakeupSource`
	`)`	

Service ID[hex]:	0x11	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	WakeupSource	Source device, which initiated the wake up event: CAN controller or CAN transceiver
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Will be returned, if the check wake up request has been accepted E_NOT_OK: Will be returned, if the check wake up request has not been accepted
Description:	This service checks, whether an underlying CAN driver or a CAN transceiver driver already signals a wakeup event.	

⌋()

Note: Integration Code calls this function

[CANIF398] ⌈If parameter `WakeupSource` of `CanIf_CheckWakeup()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_WAKEUPSOURCE` to the `Det_ReportError` service of the DET, when `CanIf_CheckWakeup()` is called.⌋(BSW00323)

[CANIF401] ⌈Caveats of `CanIf_CheckWakeup()`:

- The call context is either on interrupt level (interrupt mode) or on task level (polling mode).
- The CanIf must be initialized after Power ON.⌋()

[CANIF402] ⌈Configuration of `CanIf_CheckWakeup()`: This wake-up service is configurable by `CANIF_CTRL_WAKEUP_SUPPORT` (see CANIF637_Conf) and/or `CANIF_TRCV_WAKEUP_SUPPORT` (see CANIF606_Conf), which depends on the used CAN controller / transceiver type and the used wake-up strategy. This function may not be supported, if no wake-up shall be used.⌋()

8.3.18 CanIf_CheckValidation

[CANIF178]⌈

Service name:	CanIf_CheckValidation

Syntax:	`Std_ReturnType CanIf_CheckValidation(`
	`    EcuM_WakeupSourceType WakeupSource`
	`)`

Service ID[hex]:	0x12
Sync/Async:	Synchronous

Reentrancy:	Reentrant	
Parameters (in):	WakeupSource	Source device which initiated the wake-up event and which has to be validated: CAN controller or CAN transceiver
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Will be returned, if the check validation request has been accepted. E_NOT_OK: Will be returned, if the check validation request has not been accepted.
Description:	This service is performed to validate a previous wakeup event.	

⌋()

Note: Integration Code calls this function

[CANIF404] ⌈If parameter `WakeupSource` of `CanIf_CheckValidation()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_WAKEUPSOURCE` to the `Det_ReportError` service of the DET module, when `CanIf_CheckValidation()` is called.⌋(BSW00323)

[CANIF407] ⌈Caveats of `CanIf_CheckValidation()`:

- The CAN Interface module must be initialized after Power `ON`.
- The call context is either on interrupt level (interrupt mode) or on task level (polling mode).
- The corresponding CAN controller and transceiver must be switched on via `CanTrcv_SetOpMode(Transceiver, CANTRCV_TRCVMODE_NORMAL)` and `Can_SetControllerMode(Controller, CAN_T_START)`and the corresponding mode indications must have been called.⌋()

[CANIF408] ⌈Configuration of `CanIf_CheckValidation()`: If no validation is needed, this API can be omitted by disabling of `CANIF_PUBLIC_WAKEUP_CHECK_VALIDATION_SUPPORT` (see CANIF611_Conf)`.`
⌋()

8.3.19 CanIf_GetTxConfirmationState

[CANIF734] ⌈

Service name:	CanIf_GetTxConfirmationState	

Syntax:	`CanIf_NotifStatusType CanIf_GetTxConfirmationState(`
	`    uint8 ControllerId`
	`)`	

Service ID[hex]:	0x19	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant (Not for the same controller)	
Parameters (in):	ControllerId	Abstracted CanIf ControllerId which is assigned to a CAN controller
Parameters (inout):	None	
Parameters (out):	None	

Return value:	CanIf_NotifStatusType	Combined TX confirmation status for all TX PDUs of the CAN controller
Description:	This service reports, if any TX confirmation has been done for the whole CAN controller since the last CAN controller start.	

⌋()

[CANIF736] ⌈If parameter `ControllerId` of `CanIf_GetTxConfirmationState()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_CONTROLLERID` to the `Det_ReportError` service of the DET module, when `CanIf_GetTxConfirmationState()` is called.⌋()

[CANIF737] ⌈Caveats of `CanIf_GetTxConfirmationState()`:

- The call context is on task level (polling mode).
- The CanIf must be initialized after Power ON.⌋()

[CANIF738] ⌈Configuration of `CanIf_GetTxConfirmationState()`: If BusOff Recovery of CanSm doesn't need the status of the Tx confirmations (see CANIF740), this API can be omitted by disabling of `CANIF_PUBLIC_TXCONFIRM_POLLING_SUPPORT` (see CANIF733_Conf).⌋()

8.3.20 CanIf_ClearTrcvWufFlag

[CANIF760] ⌈

Service name:	CanIf_ClearTrcvWufFlag	

Syntax:	`Std_ReturnType CanIf_ClearTrcvWufFlag(`
	`    uint8 TransceiverId`
	`)`	

Service ID[hex]:	0x1e	
Sync/Async:	Asynchronous	
Reentrancy:	Reentrant for different CAN transceivers	
Parameters (in):	TransceiverId	designated CAN transceiver
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Request has been accepted E_NOT_OK: Request has not been accepted
Description:	Requests the CanIf module to clear the WUF flag of the designated CAN transceiver.	

⌋()

[CANIF766] ⌈ Within `CanIf_ClearTrcvWufFlag()` the function `CanTrcv_`ClearTrcvWufFlag`()` shall be called.⌋()

[CANIF769] ⌈If parameter `TransceiverId` of `CanIf_ClearTrcvWufFlag()` has an invalid value, the CanIf shall report development error code

CANIF_E_PARAM_TRCV to the Det_ReportError service of the DET module, when CanIf_ClearTrcvWufFlag() is caled.⌋()

[CANIF771] ⌈Configuration of CanIf_ClearTrcvWufFlag(): Whether the CanIf supports this function shall be pre compile time configurable On/Off by the configuration parameter CANIF_PUBLIC_PN_SUPPORT (see CANIF772_Conf).⌋()

8.3.21 CanIf_CheckTrcvWakeFlag

[CANIF761] ⌈

Service name:	CanIf_CheckTrcvWakeFlag	
Syntax:	Std_ReturnType CanIf_CheckTrcvWakeFlag(uint8 TransceiverId)	
Service ID[hex]:	0x1f	
Sync/Async:	Asynchronous	
Reentrancy:	Reentrant for different CAN transceivers	
Parameters (in):	TransceiverId	designated CAN transceiver
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Request has been accepted E_NOT_OK: Request has not been accepted
Description:	Requests the CanIf module to check the Wake flag of the designated CAN transceiver.	

⌋()

[CANIF765] ⌈ Within CanIf_CheckTrcvWakeFlag() the function CanTrcv_CheckTrcvWakeFlag() shall be called.⌋()

[CANIF770] ⌈If parameter TransceiverId of CanIf_CheckTrcvWakeFlag() has an invalid value, the CanIf shall report development error code CANIF_E_PARAM_TRCV to the Det_ReportError service of the DET module, when CanIf_CheckTrcvWakeFlag() is caled.⌋()

[CANIF813] ⌈Configuration of CanIf_CheckTrcvWakeFlag(): Whether the CanIf supports this function shall be pre compile time configurable On/Off by the configuration parameter CANIF_PUBLIC_PN_SUPPORT (see CANIF772_Conf).⌋()

8.3.22 CanIf_CheckBaudrate

[CANIF775] ⌈

Service name:	CanIf_CheckBaudrate
Syntax:	Std_ReturnType CanIf_CheckBaudrate(uint8 ControllerId,

	`    const uint16 Baudrate`
	`)`	

Service ID[hex]:	0x1a	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	ControllerId	CAN Controller to check for the support of a certain baudrate
	Baudrate	Baudrate to check in kbps
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Baudrate supported by the CAN Controller E_NOT_OK: Baudrate not supported / invalid CAN controller
Description:	This service shall check, if a certain CAN controller supports a requested baudrate	

⌋()

[CANIF786] ⌈ The service `CanIf_CheckBaudrate()` shall call `Can_CheckBaudrate(Controller, Baudrate)` for the requested CAN controller. ⌋()

[CANIF778] ⌈If parameter `ControllerId` of `CanIf_CheckBaudrate()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_CONTROLLERID` to the `Det_ReportError` service of the DET module, when `CanIf_CheckBaudrate()` is called.⌋()

Note: The parameter `Baudrate` of `CanIf_CheckBaudrate()` is not checked in CanIf. This has to be done by CAN Driver module.

[CANIF779] ⌈Caveats of `CanIf_CheckBaudrate()`:

- The call context is on task level (polling mode).
- The CanIf must be initialized after Power ON.⌋()

[CANIF780] ⌈ Configuration of `CanIf_CheckBaudrate()`: If CanIf supports changing of the baudrate and thus this service, shall be configurable via `CANIF_PUBLIC_CHANGE_BAUDRATE_SUPPORT` (see CANIF785_Conf). ⌋()

8.3.23 CanIf_ChangeBaudrate

[CANIF776]

Service name:	CanIf_ChangeBaudrate	

Syntax:	`Std_ReturnType CanIf_ChangeBaudrate(`
	`    uint8 ControllerId,`
	`    const uint16 Baudrate`
	`)`	

Service ID[hex]:	0x1b	
Sync/Async:	Asynchronous	
Reentrancy:	Reentrant	
Parameters (in):	ControllerId	CAN Controller, whose baudrate shall be changed

	Baudrate	Requested baudrate in kbps
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Service request accepted, baudrate change started E_NOT_OK: Service request not accepted
Description:	This service shall change the baudrate of the CAN controller.	

⌋()

[CANIF787] ⌈ The service `CanIf_ChangeBaudrate()` shall call `Can_ChangeBaudrate(Controller, Baudrate)` for the requested CAN controller. ⌋()

[CANIF782] ⌈If parameter `ControllerId` of `CanIf_ChangeBaudrate()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_CONTROLLERID` to the `Det_ReportError` service of the DET module, when `CanIf_ChangeBaudrate()` is called.⌋()

Note: The parameter `Baudrate` of `CanIf_ChangeBaudrate()` is not checked in CanIf. This has to be done by CAN Driver module.

[CANIF783] ⌈Caveats of `CanIf_ChangeBaudrate()`:

- The call context is on task level (polling mode).
- The CanIf must be initialized after Power ON.⌋()

[CANIF784] ⌈ Configuration of `CanIf_ChangeBaudrate()`:If CanIf supports changing of the baudrate and thus this service, shall be configurable via `CANIF_PUBLIC_CHANGE_BAUDRATE_SUPPORT` (see CANIF785_Conf). ⌋()

8.4 Callback notifications

This is a list of functions provided for other modules.

[CANIF409] ⌈The function prototypes of the CAN Interface module's callback functions shall be provided in the file `CanIf_Cbk.h`.⌋()

Note: This callback service in this chapter are implemented as many times as underlying CAN Driver modules are used. In that case one callback is assigned to one underlying CAN Driver module. The following naming convention has to be considered: `CanIf_<Callback function>_<CAN_Driver>` (See [CANIF124in subchapter 7.25.)

8.4.1 CanIf_TxConfirmation

[CANIF007] ⌈

Service name:	CanIf_TxConfirmation	

Syntax:	`void CanIf_TxConfirmation(`
	`    PduIdType CanTxPduId`
	`)`	

Service ID[hex]:	0x13	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	CanTxPduId	L-PDU handle of CAN L-PDU successfully transmitted. This ID specifies the corresponding CAN L-PDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service confirms a previously successfully processed transmission of a CAN TxPDU.	

⌋(BSW01009)

Note: The service `CanIf_TxConfirmation()` is implemented in the CAN Interface module and called by the CAN Driver module after the CAN L-PDU has been transmitted on the CAN network.

Note: Within the service `CanIf_TxConfirmation()`, the CAN Driver module passes back the `CanTxPduId` to the CAN Interface module, which it got from `Can_Write(Hth, *PduInfo).`

.**[CANIF391]** ⌈ If configuration parameters `CANIF_PUBLIC_READTXPDU_NOTIFY_STATUS_API` (CANIF609_Conf) and `CANIF_TXPDU_READ_NOTIFYSTATUS` (CANIF589_Conf for the transmitted L-PDU are set to TRUE, and if `CanIf_TxConfirmation()` is called, the CanIf shall set the notification status for the transmitted L-PDU. ⌋()

[CANIF410] ⌈If parameter `CanTxPduId` of `CanIf_TxConfirmation()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_LPDU` to the `Det_ReportError` service of the DET module, when `CanIf_TxConfirmation()` is called.⌋(BSW00323)

[CANIF412] ⌈If the CanIf was not initialized before calling `CanIf_TxConfirmation()`, the CanIf shall not call the service `<User_TxConfirmation>()` and shall not set the Tx confirmation status, when `CanIf_TxConfirmation()` is called.⌋()

[CANIF413] ⌈Caveats of `CanIf_TxConfirmation()`:

- The call context is either on interrupt level (interrupt mode) or on task level (polling mode).
- The CanIf must be initialized after Power ON.⌋()

[CANIF414] ⌈Configuration of `CanIf_TxConfirmation()`: Each CAN Tx L-PDU (see CANIF248_Conf) has to be configured with a corresponding transmit confirmation service of an upper layer module (see CANIF011) which is called in `CanIf_TxConfirmation()`.⌋()

8.4.2 CanIf_RxIndication

[CANIF006] ⌈

Service name:	CanIf_RxIndication	

Syntax:	`void CanIf_RxIndication(`
	`    Can_HwHandleType Hrh,`
	`    Can_IdType CanId,`
	`    uint8 CanDlc,`
	`    const uint8* CanSduPtr`
	`)`	

Service ID[hex]:	0x14	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	Hrh	ID of the corresponding Hardware Object Range: 0..(total number of HRH -1)
	CanId	Standard/Extended CAN ID of CAN L-PDU that has been successfully received
	CanDlc	Data Length Code (length of CAN L-PDU payload)
	CanSduPtr	Pointer to received L-SDU (payload)
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service indicates a successful reception of a received CAN Rx L-PDU to the CanIf after passing all filters and validation checks.	

⌋()

Note: The service CanIf_RxIndication() is implemented in the CAN Interface module and called by the CAN Driver module after a CAN L-PDU has been received.

[CANIF415] ⌈Within the service `CanIf_RxIndication()` the CAN Interface module translates the `CanId` into the configured target PDU ID and routes this indication to the configured upper layer target service(s).⌋()

[CANIF392] ⌈ If configuration parameters `CANIF_PUBLIC_READRXPDU_NOTIFY_STATUS_API` (CANIF608_Conf) and `CANIF_RXPDU_READ_NOTIFYSTATUS` (CANIF595_Conf) for the received L-PDU are set to TRUE, and if `CanIf_RxIndication()` is called, the CanIf shall set the notification status for the received L-PDU.⌋()

[CANIF416] ⌈If parameter Hrh of CanIf_RxIndication() has an invalid value, the CanIf shall report development error code CANIF_E_PARAM_HRH to the Det_ReportError service of the DET module, when CanIf_RxIndication() is called.⌋(BSW00323)

[CANIF417] ⌈If parameter CanId of CanIf_RxIndication() has an invalid value, the CanIf shall report development error code CANIF_E_PARAM_CANID to the Det_ReportError service of the DET module, when CanIf_RxIndication() is called.⌋(BSW00323)

[CANIF418] ⌈If parameter CanDlc of CanIf_RxIndication() has an invalid value, the CanIf shall report development error code CANIF_E_PARAM_DLC to the Det_ReportError service of the DET module, when CanIf_RxIndication() is called.⌋(BSW00323)

[CANIF419] ⌈If parameter CanSduPtr of CanIf_RxIndication() has an invalid value, the CanIf shall report development error code CANIF_E_PARAM_POINTER to the Det_ReportError service of the DET module, when CanIf_RxIndication() is called.⌋(BSW00323)

[CANIF421] ⌈ If the CanIf was not initialized before calling CanIf_RxIndication(), the CanIf shall not execute Rx indication handling, when CanIf_RxIndication(), is called.⌋()

[CANIF422] ⌈Caveats of CanIf_RxIndication():

- The call context is either on interrupt level (interrupt mode) or on task level (polling mode).
- The CanIf must be initialized after Power ON.⌋()

[CANIF423] ⌈Configuration of CanIf_RxIndication(): Each CAN Rx L-PDU (see CANIF249_Conf) has to be configured with a corresponding receive indication service of an upper layer module (see CANIF012) which is called in CanIf_RxIndication().⌋()

8.4.3 CanIf_CancelTxConfirmation

[CANIF101] ⌈

Service name:	CanIf_CancelTxConfirmation
Syntax:	void CanIf_CancelTxConfirmation(PduIdType CanTxPduId, const PduInfoType* PduInfoPtr)

Service ID[hex]:	0x15	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	CanTxPduId	ID of the L-PDU which shall be buffered in CanIf and replaced by a new pending L-PDU with a higher priority.
	PduInfoPtr	Pointer to struct which contains the address of the HTH in which the L-PDU is located and the length of the L-PDU.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service informs CanIf that a L-PDU shall be buffered in CanIf Tx´buffer from CAN hardware object to avoid priority inversion.	

⌋()

Note: The service `CanIf_CancelTxConfirmation()` is implemented in the CanIf and called by the CanDrv after a previous request for cancellation of a pending L-PDU transmit request was successfully performed.

[CANIF424] ⌈If parameter `CanTxPduId` of `CanIf_CancelTxConfirmation()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_LPDU` to the `Det_ReportError` service of the DET module, when `CanIf_CancelTxConfirmation()` is called.⌋(BSW00323)

[CANIF828] ⌈If parameter `PduInfoPtr` of `CanIf_CancelTxConfirmation()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_POINTER` to the `Det_ReportError` service of the DET module, when `CanIf_CancelTxConfirmation()` is called.⌋(BSW00323)

[CANIF426] ⌈ If the CanIf was not initialized before calling `CanIf_CancelTxConfirmation()`,the CanIf shall not execute Tx cancellation handling, when `CanIf_CancelTxConfirmation()` is called.⌋()

[CANIF427] ⌈Caveats of `CanIf_CancelTxConfirmation()`:

- The call context is either on interrupt level (interrupt mode) or on task level (polling mode).
- The CanIf must be initialized after Power ON.⌋()

[CANIF428] ⌈Configuration of `CanIf_CancelTxConfirmation()`: This function shall be pre compile time configurable `On/Off` by the configuration parameter `CANIF_CTRLDRV_TX_CANCELLATION` (see CANIF640_Conf).⌋()

8.4.4 CanIf_ControllerBusOff

[CANIF218] ⌈

Service name:	CanIf_ControllerBusOff	

Syntax:	`void CanIf_ControllerBusOff(`
	`    uint8 ControllerId`
	`)`	

Service ID[hex]:	0x16	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	ControllerId	CAN controller, where a BusOff occured
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service indicates a Controller BusOff event referring to the corresponding CAN Controller.	

⌋()

Note: The callback service `CanIf_ControllerBusOff()` is called by the CanDrv and implemented in the CanIf. It is called in case of a mode change notification of the CanDrv.

[CANIF429] ⌈If parameter `ControllerId` of `CanIf_ControllerBusOff()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_CONTROLLER` to the `Det_ReportError` service of the DET module, when `CanIf_ControllerBusOff()` is called.⌋(BSW00323)

[CANIF431] ⌈ If the CanIf was not initialized before calling `CanIf_ControllerBusOff()`, the CanIf shall not execute BusOff notification, when `CanIf_ControllerBusOff()`, is called.⌋()

[CANIF432] ⌈Caveats of `CanIf_ControllerBusOff()`:

- The call context is either on interrupt level (interrupt mode) or on task level (polling mode).
- The CanIf must be initialized after Power ON.⌋()

[CANIF433] ⌈Configuration of `CanIf_ControllerBusOff()`: ID of the CAN controller is published inside the configuration description of the CanIf (see CANIF546_Conf).⌋()

Note: This service always has to be available, so there does not exist an appropriate configuration parameter.

8.4.5 CanIf_ConfirmPnAvailability

[CANIF815] ⌈

Service name:	CanIf_ConfirmPnAvailability	

Syntax:	`void CanIf_ConfirmPnAvailability(`
	`    uint8 TransceiverId`
	`)`	

Service ID[hex]:	0x1a	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	TransceiverId	CAN transceiver, which was checked for PN availability
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service indicates that the transceiver is running in PN communication mode.	

⌋()

[CANIF753]⌈If `CanIf_ConfirmPnAvailability()` is called, the CanIf calls `<User_ConfirmPnAvailability>()`.⌋()

Note: The CanIf passes the delivered parameter `TransceiverId` to the upper layer module.

[CANIF816] ⌈ If parameter `TransceiverId` of `CanIf_ConfirmPnAvailability()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_TRANSCEIVER` to the `Det_ReportError` service of the DET module, when `CanIf_ConfirmPnAvailability()` is called.⌋()

[CANIF817] ⌈ If the CanIf was not initialized before calling `CanIf_ConfirmPnAvailability()`, the CanIf shall not execute notification, when `CanIf_ConfirmPnAvailability()` is called.⌋()

[CANIF818]⌈Caveats of `CanIf_ConfirmPnAvailability()`:

- The call context is either on interrupt level (interrupt mode) or on task level (polling mode).
- The CanIf must be initialized after Power ON.⌋()

[CANIF754] ⌈Configuration of `CanIf_ConfirmPnAvailability()`:This function shall be pre compile time configurable `On/Off` by the configuration parameter `CANIF_PUBLIC_PN_SUPPORT` (see CANIF772_Conf).⌋()

8.4.6 CanIf_ClearTrcvWufFlagIndication

[CANIF762] ⌈

Service name:	CanIf_ClearTrcvWufFlagIndication

Syntax:	`void CanIf_ClearTrcvWufFlagIndication(`
	`    uint8 TransceiverId`

	)	
Service ID[hex]:	0x20	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	TransceiverId	CAN transceiver, for which this function was called.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service indicates that the transceiver has cleared the WufFlag.	

⌋()

[CANIF757] ⌈If `CanIf_ClearTrcvWufFlagIndication()` is called, the CanIf calls <`User_ClearTrcvWufFlagIndication`>`()`.⌋()

Note: The CanIf passes the delivered parameter `TransceiverId` to the upper layer module.

[CANIF805] ⌈ If parameter `TransceiverId` of `CanIf_ClearTrcvWufFlagIndication()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_TRANSCEIVER` to the `Det_ReportError` service of the DET module, when `CanIf_ClearTrcvWufFlagIndication()` is called.⌋()

[CANIF806] ⌈ If the CanIf was not initialized before calling `CanIf_ClearTrcvWufFlagIndication()`, the CanIf shall not execute notification, when `CanIf_ClearTrcvWufFlagIndication()` is called.⌋()

[CANIF807] ⌈Caveats of `CanIf_ClearTrcvWufFlagIndication()`:

- The call context is either on interrupt level (interrupt mode) or on task level (polling mode).
- The CanIf must be initialized after Power ON.⌋()

[CANIF808] ⌈ Configuration of `CanIf_ClearTrcvWufFlagIndication()`:This function shall be pre compile time configurable `On/Off` by the configuration parameter `CANIF_PUBLIC_PN_SUPPORT` (see CANIF772_Conf).⌋()

8.4.7 CanIf_CheckTrcvWakeFlagIndication

[CANIF763] ⌈

Service name:	CanIf_CheckTrcvWakeFlagIndication

Syntax:	`void CanIf_CheckTrcvWakeFlagIndication(`
	`    uint8 TransceiverId`
	`)`

Service ID[hex]:	0x21
Sync/Async:	Synchronous

Reentrancy:	Reentrant	
Parameters (in):	TransceiverId	CAN transceiver, for which this function was called.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service indicates the reason for the wake up that the CAN transceiver has detected.	

⌋()

[CANIF759] ⌈If `CanIf_CheckTrcvWakeFlagIndication()` is called, the CanIf calls `<User_CheckTrcvWakeFlagIndication>()`.⌋()

Note: The CanIf passes the delivered parameter `TransceiverId` to the upper layer module.

[CANIF809] ⌈ If parameter `TransceiverId` of `CanIf_CheckTrcvWakeFlagIndication()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_TRANSCEIVER` to the `Det_ReportError` service of the DET module, when `CanIf_CheckTrcvWakeFlagIndication()` is called.⌋()

[CANIF810] ⌈ If the CanIf was not initialized before calling `CanIf_CheckTrcvWakeFlagIndication()`, the CanIf shall not execute notification, when `CanIf_CheckTrcvWakeFlagIndication()` is called.⌋()

[CANIF811] ⌈Caveats of `CanIf_CheckTrcvWakeFlagIndication()`:

- The call context is either on interrupt level (interrupt mode) or on task level (polling mode).
- The CanIf must be initialized after Power ON.⌋()

[CANIF812] ⌈Configuration of `CanIf_CheckTrcvWakeFlagIndication()`:This function shall be pre compile time configurable `On/Off` by the configuration parameter `CANIF_PUBLIC_PN_SUPPORT` (see CANIF772_Conf).⌋()

8.4.8 CanIf_ControllerModeIndication

[CANIF699] ⌈

Service name:	CanIf_ControllerModeIndication

Syntax:	`void CanIf_ControllerModeIndication(`
	`    uint8 ControllerId,`
	`    CanIf_ControllerModeType ControllerMode`
	`)`

Service ID[hex]:	0x17
Sync/Async:	Synchronous
Reentrancy:	Reentrant

Parameters (in):	ControllerId	CAN controller, which state has been transitioned.
	ControllerMode	Mode to which the CAN controller transitioned
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service indicates a controller state transition referring to the corresponding CAN controller.	

⌋()

Note: The callback service `CanIf_ControllerModeIndication()` is called by the CanDrv and implemented in the CanIf. It is called in case of a state transition notification of the CanDrv.

[CANIF700] ⌈ If parameter `ControllerId` of `CanIf_ControllerModeIndication()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_CONTROLLER` to the `Det_ReportError` service of the DET module, when `CanIf_ControllerModeIndication()` is called.⌋()

[CANIF702] ⌈ If the CanIf was not initialized before calling `CanIf_ControllerModeIndication()`, the CanIf shall not execute state transition notification, when `CanIf_ControllerModeIndication()` is called.⌋()

[CANIF703] ⌈Caveats of `CanIf_ControllerModeIndication()`:

- The call context is either on interrupt level (interrupt mode) or on task level (polling mode).
- The CanIf must be initialized after Power ON.⌋()

[CANIF704] ⌈Configuration of `CanIf_ControllerModeIndication()`: ID of the CAN controller is published inside the configuration description of the CanIf (see CANIF647_Conf).⌋()

8.4.9 CanIf_TrcvModeIndication

[CANIF764] ⌈

Service name:	CanIf_TrcvModeIndication	

Syntax:	`void CanIf_TrcvModeIndication(`
	`    uint8 TransceiverId,`
	`    CanTrcv_TrcvModeType TransceiverMode`
	`)`	

Service ID[hex]:	0x18	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	TransceiverId	CAN transceiver, which state has been transitioned.
	TransceiverMode	Mode to which the CAN transceiver transitioned
Parameters	None	

(inout):	
Parameters (out):	None
Return value:	None
Description:	This service indicates a transceiver state transition referring to the corresponding CAN transceiver.

⌋()

Note: The callback service `CanIf_TrcvModeIndication()` is called by the CanDrv and implemented in the CanIf. It is called in case of a state transition notification of the CanDrv.

[CANIF706] ⌈If parameter `TransceiverId` of `CanIf_TrcvModeIndication()` has an invalid value, the CanIf shall report development error code `CANIF_E_PARAM_TRCV` to the `Det_ReportError` service of the DET module, when `CanIf_TrcvModeIndication()` is called.⌋()

[CANIF708] ⌈ If the CanIf was not initialized before calling `CanIf_TrcvModeIndication()`, the CanIf shall not execute state transition notification, when `CanIf_TrcvModeIndication()` is called. ⌋()

[CANIF709] ⌈Caveats of `CanIf_TrcvModeIndication()`:

- The call context is either on interrupt level (interrupt mode) or on task level (polling mode).
- The CanIf must be initialized after Power ON.⌋()

[CANIF710] ⌈Configuration of `CanIf_TrcvModeIndication()`: ID of the CAN transceiver is published inside the configuration description of the CanIf via parameter CANIF_TRCV_ID (see CANIF654_Conf). ⌋()

[CANIF730] ⌈Configuration of `CanIf_TrcvModeIndication ()`: If transceivers are not supported (`CanIfTrcvDrvCfg` is not configured, see CANIF273_Conf), `CanIf_TrcvModeIndication()` shall not be provided by CanIf.⌋()

8.5 Scheduled functions

Note: The CAN Interface module does not have scheduled functions or needs some.

8.6 Expected interfaces

In this chapter all interfaces required from other modules are listed.

8.6.1 Mandatory interfaces

Note: This chapter defines all interfaces, which are required to fulfill the core functionality of the module.

[CANIF040] ⌈

API function	***Description***
Can_SetControllerMode	This function performs software triggered state transitions of the CAN controller State machine.
Can_Write	--
SchM_Enter_CanIf_<ExclusiveArea>	Invokes the SchM_Enter function to enter a module local exclusive area.
SchM_Exit_CanIf_<ExclusiveArea>	Invokes the SchM_Exit function to exit an exclusive area.

⌋()

8.6.2 Optional interfaces

This chapter defines all interfaces, which are required to fulfill an optional functionality of the module.

[CANIF294] ⌈

API function	***Description***
CanTrcv_CheckWakeup	Service is called by underlying CANIF in case a wake up interrupt is detected.
CanTrcv_GetBusWuReason	Gets the wakeup reason for the Transceiver and returns it in parameter Reason.
CanTrcv_GetOpMode	Gets the mode of the Transceiver and returns it in OpMode.
CanTrcv_SetOpMode	Sets the mode of the Transceiver to the value OpMode.
CanTrcv_SetWakeupMode	Enables, disables or clears wake-up events of the Transceiver according to TrcvWakeupMode.
Can_ChangeBaudrate	This service shall change the baudrate of the CAN controller.
Can_CheckBaudrate	This service shall check, if a certain CAN controller supports a requested baudrate
Can_CheckWakeup	This function checks if a wakeup has occurred for the given controller.
Det_ReportError	Service to report development errors.

⌋()

8.6.3 Configurable interfaces

In this chapter all interfaces are listed, where the target function of any upper layer to be called has to be set up by configuration. These callback services are specified and implemented in the upper communication modules, which use the CAN Interface according to the AUTOSAR BSW architecture. The specific callback notification is specified in the corresponding SWS document (see chapter [3 Related documentation]).

As far the interface name is not specified to be mandatory, no callback is performed, if no API name is configured. This chapter describes only the content of notification of the callback, the call context inside the CanIf and exact time by the call event.

<User_NotificationName> - This condition is applied for such interface services which will be implemented in the upper layer and called by the CAN Interface module. This condition displays the symbolic name of the functional group in a callback service in the corresponding upper layer module. Each upper layer module can define no, one or several callback services for the same functionality (i.e. transmit confirmation). The dispatch is ensured by the L-PDU ID.

The upper layer module provides the Service ID of the following functions.

8.6.3.1 <User_TxConfirmation>

[CANIF011] ⌈

Service name:	<User_TxConfirmation>	

Syntax:	`void <User_TxConfirmation>(`
	`    PduIdType TxPduId`
	`)`	

Sync/Async:	Synchronous	
Reentrancy:	Reentrant for different PduIds. Non reentrant for the same PduId.	
Parameters (in):	TxPduId	ID of the I-PDU that has been transmitted.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	The lower layer communication module confirms the transmission of an I-PDU.	

⌋()

Note: This callback service is called by the CanIf and implemented in the corresponding upper layer module. It is called in case of a transmit confirmation of the CanDrv.

Note: This type of confirmation callback service is mainly designed for the PduR, CanNm and CanTp, but not exclusive.

Note: Parameter TxPduId specifies the corresponding CAN L-PDU ID and implicitly the CanDrv instance as well as the corresponding CAN controller device. The range is between 0 and ((maximum number of L-PDU IDs which may be transmitted by the CanIf) -1).

[CANIF437] ⌈Caveats of `<User_TxConfirmation>()`: The call context is either on interrupt level (interrupt mode) or on task level (polling mode).⌋()

Note: This kind of callback function is in general re-entrant for multiple CAN controller or multiple CAN network usage (for different L-PDU IDs), but not for the same CAN controller or CAN network (the same L-PDU ID).

[CANIF438] ⌈Configuration of `<User_TxConfirmation>()`: The upper layer module, which provides this callback service, has to be configured by `CANIF_TXPDU_USERTXCONFIRMATION_UL` (see CANIF527_Conf). If no upper layer modules are configured for transmit confirmation using `<User_TxConfirmation>()`, no transmit confirmation is executed.⌋()

[CANIF542] ⌈Configuration of <User_TxConfirmation>(): The name of the API <User_TxConfirmation>() which is called by CanIf shall be configured for the CanIf by parameter CANIF_TXPDU_USERTXCONFIRMATION_NAME (see CANIF528_Conf).⌋()

Note: If transmit confirmations are not necessary or no upper layer modules are configured for transmit confirmations and thus <User_TxConfirmation>() shall not be called, CANIF_TXPDU_USERTXCONFIRMATION_UL and CANIF_TXPDU_USERTXCONFIRMATION_NAME need not to be configured.

[CANIF439] ⌈Configuration of <User_TxConfirmation>(): If CANIF_TXPDU_USERTXCONFIRMATION_UL is set to PDUR, the following is prescribed:

- CANIF_TXPDU_USERTXCONFIRMATION_NAME must be PduR_CanIfTxConfirmation
- function parameter of type PduIdType has to be named as CanTxPduId⌋()

[CANIF543] ⌈Configuration of <User_TxConfirmation>(): If CANIF_TXPDU_USERTXCONFIRMATION_UL is set to CAN_NM, the following is prescribed:

- CANIF_TXPDU_USERTXCONFIRMATION_NAME must be CanNm_TxConfirmation
- function parameter of type PduIdType has to be named as canNmTxPduId⌋()

Hint (Dependency to another module):
If at least one CanIf Tx L-PDU is configured with CanNm_TxConfirmation(), which means CANIF_TXPDU_USERTXCONFIRMATION_UL equals CAN_NM, the CanNm configuration parameter CANNM_IMMEDIATE_TXCONF_ENABLED must be set to FALSE (see [12] Specification of CAN Network Management, CANNM284).

[CANIF544] ⌈Configuration of <User_TxConfirmation>(): If CANIF_TXPDU_USERTXCONFIRMATION_UL is set to J1939TP, the following is prescribed:

- CANIF_TXPDU_USERTXCONFIRMATION_NAME must be J1939Tp_TxConfirmation
- function parameter of type PduIdType has to be named as J1939TpTxPduId⌋()

[CANIF550] ⌈Configuration of <User_TxConfirmation>(): If CANIF_TXPDU_USERTXCONFIRMATION_UL is set to CAN_TP, the following is prescribed:

- CANIF_TXPDU_USERTXCONFIRMATION_NAME must be CanTp_TxConfirmation
- function parameter of type PduIdType has to be named as CanTpTxPduId⌋()

[CANIF556] ⌈Configuration of `<User_TxConfirmation>()`: If `CANIF_TXPDU_USERTXCONFIRMATION_UL` is set to `XCP`, the following is prescribed:

- `CANIF_TXPDU_USERTXCONFIRMATION_NAME` must be `Xcp_CanIfTxConfirmation`
- function parameter of type `PduIdType` has to be named as `XcpTxPduId`⌋()

[CANIF551] ⌈Configuration of `<User_TxConfirmation>()`: If `CANIF_TXPDU_USERTXCONFIRMATION_UL` is set to `CDD`, the name of the API `<User_TxConfirmation>()`has to be configured via parameter `CANIF_TXPDU_USERTXCONFIRMATION_NAME`. The function parameter has to be of type `PduIdType`.⌋()

8.6.3.2 <User_RxIndication>

[CANIF012] ⌈

Service name:	<User_RxIndication>	

Syntax:	`void <User_RxIndication>(`
	`    PduIdType RxPduId,`
	`    PduInfoType* PduInfoPtr`
	`)`	

Sync/Async:	Synchronous	
Reentrancy:	Reentrant for different PduIds. Non reentrant for the same PduId.	
Parameters (in):	RxPduId	ID of the received I-PDU.
	PduInfoPtr	Contains the length (SduLength) of the received I-PDU and a pointer to a buffer (SduDataPtr) containing the I-PDU.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	Indication of a received I-PDU from a lower layer communication module.	

⌋(BSW01003)

Note: This service indicates a successful reception of an L-PDU to the upper layer module after passing all filters and validation checks.

Note: This callback service is called by the CanIf and implemented in the configured upper layer module (e.g. PduR, CanNm, CanTp, etc.) if configured accordingly (see CANIF529_Conf).

Note: Parameter / handle RxPduId identifies the received data. The range is between 0 and ((maximum number of L-PDU IDs which may be received by the CanIf) -1).

[CANIF440] ⌈Caveats of `<User_RxIndication>`:

- Until this service returns, the CanIf will not access `<PduInfoPtr>`. The `<PduInfoPtr>` is only valid and can be used by upper layers, until the indication returns. The CanIf guarantees that the number of configured bytes for this `<PduInfoPtr>` is valid.

- The CAN Driver module must be initialized after Power ON.
- The call context is either on interrupt level (interrupt mode) or on task level (polling mode).⌋()

Note: This kind of callback function is in general reentrant for multiple CAN controller or multiple CAN network usage (for different L-PDU IDs), but not for the same CAN controller or CAN network (the same L-PDU ID).

[CANIF441] ⌈Configuration of `<User_RxIndication>()`: The upper layer module, which provides this callback service, has to be configured by `CANIF_RXPDU_USERRXINDICATION_UL` (see CANIF529_Conf).⌋()

[CANIF552] ⌈Configuration of `<User_RxIndication>()`: The name of the API `<User_RxIndication>()` which will be called by the CanIf shall be configured for the CanIf by parameter `CANIF_RXPDU_USERRXINDICATION_NAME` (see CANIF530_Conf).⌋()

Note: If receive indications are not necessary or no upper layer modules are configured for receive indications and thus `<User_RxIndication>`() shall not be called, `CANIF_RXPDU_USERRXINDICATION_UL` and `CANIF_RXPDU_USERRXINDICATION_NAME` need not to be configured.

[CANIF442] ⌈Configuration of `<User_RxIndication>()`: If `CANIF_RXPDU_USERRXINDICATION_UL` is set to `PDUR`, the following is prescribed:

- `CANIF_RXPDU_USERRXINDICATION_NAME` must be `PduR_CanIfRxIndication`
- function parameter of type `PduIdType` has to be named as `id`
- function parameter of type `const PduInfoType` has to be named as `buffer`⌋()

[CANIF445] ⌈Configuration of `<User_RxIndication>()`: If `CANIF_RXPDU_USERRXINDICATION_UL` is set to `CAN_NM`, the following is prescribed:

- `CANIF_RXPDU_USERRXINDICATION_NAME` must be `CanNm_RxIndication`
- function parameter of type `PduIdType` has to be named as `CanNmRxPduId`
- function parameter of type `const PduInfoType` has to be named as `CanNmRxPduPtr`⌋()

The value passed to CanNm via the API parameter `CanNmRxPduId` refers to the CanNm channel handle within the CanNm module (see [12] Specification of CAN Network Management).

[CANIF448]⌈Configuration of `<User_RxIndication>()`: If `CANIF_RXPDU_USERRXINDICATION_UL` is set to `CAN_TP`, the following is prescribed:

- `CANIF_RXPDU_USERRXINDICATION_NAME` must be `CanTp_RxIndication`
- function parameter of type `PduIdType` has to be named as `CanTpRxPduId`
- function parameter of type `const PduInfoType` has to be named as `CanTpRxPduPtr`⌋()

[CANIF554]⌈Configuration of `<User_RxIndication>()`: If `CANIF_RXPDU_USERRXINDICATION_UL` is set to `J1939TP`, the following is prescribed:

- `CANIF_RXPDU_USERRXINDICATION_NAME` must be `J1939Tp_RxIndication`
- function parameter of type `PduIdType` has to be named as `J1939TpRxPduId`
- function parameter of type `const PduInfoType` has to be named as `J1939TpRxPduPtr`⌋()

[CANIF555]⌈Configuration of `<User_RxIndication>()`: If `CANIF_RXPDU_USERRXINDICATION_UL` is set to `XCP`, the following is prescribed:

- `CANIF_RXPDU_USERRXINDICATION_NAME` must be `Xcp_CanIfRxIndication`
- function parameter of type `PduIdType` has to be named as `XcpRxPduId`
- function parameter of type `const PduInfoType` has to be named as `XcpRxPduPtr`⌋()

[CANIF557]⌈Configuration of `<User_RxIndication>()`: If `CANIF_RXPDU_USERRXINDICATION_UL` is set to `CDD` the name of the API has to be configured via parameter `CANIF_RXPDU_USERRXINDICATION_NAME`. ⌋()

8.6.3.3 <User_ValidateWakeupEvent>

[CANIF532] ⌈

Service name:	<User_ValidateWakeupEvent>	

Syntax:	`void <User_ValidateWakeupEvent>(`
	`    EcuM_WakeupSourceType sources`
	`)`	

Sync/Async:	(defined within providing upper layer module)	
Reentrancy:	(defined within providing upper layer module)	
Parameters (in):	sources	Validated CAN wakeup events. Every CAN controller or CAN transceiver can be a separate wakeup source.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service indicates if a wake up event initiated from the wake up source (CAN controller or transceiver) after a former request to the CAN Driver or CAN Transceiver Driver module is valid.	

⌋()

Note: This callback service is mainly implemented in and used by the ECU State Manager module (see Specification of ECU State Manager [15]).

Note: The CanIf calls this callback service. It is implemented by the configured upper layer module. It is called only during the call of `CanIf_CheckValidation()`if a first CAN L_PDU reception event after a wake up event has been occurred at the corresponding CAN controller.

[CANIF455] ⌈Caveats of `<User_ValidateWakeupEvent>`:

- The CanDrv must be initialized after Power ON.
- The call context is either on interrupt level (interrupt mode) or on task level (polling mode).
- This callback service is in general re-entrant for multiple CAN controller usage, but not for the same CAN controller.⌋()

[CANIF659] ⌈Configuration of `<User_ValidateWakeupEvent>`: If no validation is needed, this API can be omitted by disabling `CANIF_PUBLIC_WAKEUP_CHECK_VALIDATION_SUPPORT` (see CANIF611_Conf).
⌋()

[CANIF456] ⌈Configuration of `<User_ValidateWakeupEvent>`: The upper layer module which provides this callback service has to be configured by `CANIF_DISPATCH_USERVALIDATEWAKEUPEVENT_UL` (see CANIF549_Conf), but:

- If no upper layer modules are configured for wake up notification using `<User_ValidateWakeupEvent>()`, no wake up notification needs to be configured. `CANIF_DISPATCH_USERVALIDATEWAKEUPEVENT_UL` needs not to be configured.
- If wake up is not supported (`CANIF_CTRL_WAKEUP_SUPPORT` and `CANIF_TRCV_WAKEUP_SUPPORT` equal FALSE, see CANIF637_Conf, CANIF606_Conf), `CANIF_DISPATCH_USERVALIDATEWAKEUPEVENT_UL` is not configurable.⌋()

[CANIF563] ⌈Configuration of <User_ValidateWakeupEvent>(): If `CANIF_DISPATCH_USERVALIDATEWAKEUPEVENT_UL` is set to ECUM, the following is prescribed:

- `CANIF_DISPATCH_USERVALIDATEWAKEUPEVENT_NAME` must be EcuM_ValidateWakeupEvent
- function parameter of type `EcuM_WakeupSourceType` has to be named as sources⌋()

[CANIF564] ⌈Configuration of `<User_ValidateWakeupEvent>()`: If `CANIF_DISPATCH_USERVALIDATEWAKEUPEVENT_UL` is set to CDD the name of the API has to be configured via parameter `CANIF_DISPATCH_USERVALIDATEWAKEUPEVENT_NAME.` The function parameter has to be of type `EcuM_WakeupSourceType.`⌋()

8.6.3.4 <User_ControllerBusOff>

[CANIF014] ⌈

Service name:	<User_ControllerBusOff>	

Syntax:	`void <User_ControllerBusOff>(`
	`    uint8 ControllerId`
	`)`	

Sync/Async:	(defined within providing upper layer module)	
Reentrancy:	(defined within providing upper layer module)	
Parameters (in):	ControllerId	Abstracted CanIf ControllerId which is assigned to a CAN controller, at which a BusOff occurred.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service indicates a bus-off event to the corresponding upper layer module (mainly the CAN State Manager module).	

⌋(BSW01029)

Note: This callback service is mainly implemented in and used by the CanSm (see Specification of CAN State Manager [11]).

Note: This callback service is called by the CanIf and implemented by the configured upper layer module. It is called in case of a BusOff notification via `CanIf_ControllerBusOff()` of the CanDrv. The delivered parameter `ControllerId` of the service `CanIf_ControllerBusOff()` is passed to the upper layer module.

[CANIF449] ⌈Caveats of `<User_ControllerBusOff>()`:

- The CanDrv must be initialized after Power ON.
- The call context is either on interrupt level (interrupt mode) or on task level (polling mode).
- This callback service is in general re-entrant for multiple CAN controller usage, but not for the same CAN controller.
- Before re-initialization/restart during BusOff recovery is executed this callback service is performed only once in case of multiple BusOff events at CAN controller.⌋()

[CANIF450] ⌈Configuration of `<User_ControllerBusOff>()`:
The upper layer module which provides this callback service has to be configured by `CANIF_DISPATCH_USERCTRLBUSOFF_UL` (see CANIF547_Conf). ⌋()

[CANIF558] ⌈Configuration of `<User_ControllerBusOff>()`: The name of the API `<User_ControllerBusOff>()` which will be called by the CanIf shall be configured for the CanIf by parameter CANIF_DISPATCH_USERCTRLBUSOFF_NAME (see CANIF525_Conf).⌋()

[CANIF524] ⌈Configuration of <User_ControllerBusOff>(): At least one upper layer module and hence an API of <User_ControllerBusOff>() has mandatorily to be configured, which the CanIf can call in case of an occurred call of CanIf_ControllerBusOff().⌋()

[CANIF559] ⌈Configuration of <User_ControllerBusOff>(): If CANIF_DISPATCH_USERCTRLBUSOFF_UL is set to CAN_SM, the following is prescribed:

- CANIF_DISPATCH_USERCTRLBUSOFF_NAME must be CanSM_ControllerBusOff
- function parameter of type uint8 has to be named as Controller⌋()

[CANIF560] ⌈Configuration of <User_ControllerBusOff>(): If CANIF_DISPATCH_USERCTRLBUSOFF_UL is set to CDD the name of the API has to be configured via parameter CANIF_DISPATCH_USERCTRLBUSOFF_NAME. The function parameter has to be of type uint8.⌋()

8.6.3.5 <User_ConfirmPnAvailability>

[CANIF821] ⌈

Service name:	<User_ConfirmPnAvailability>	
Syntax:	void <User_ConfirmPnAvailability>(uint8 TransceiverId)	
Sync/Async:	(defined within providing upper layer module)	
Reentrancy:	(defined within providing upper layer module)	
Parameters (in):	TransceiverId	CAN transceiver, which was checked for PN availability
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service indicates that the CAN transceiver is running in PN communication mode.	

⌋()

Note: This callback service is mainly implemented in and used by the CanSm (see Specification of CAN State Manager [11]).

[CANIF822] ⌈Caveats of <User_ConfirmPnAvailability>():

- The CanTrcvDrv must be initialized after Power ON.
- The call context is either on interrupt level (interrupt mode) or on task level (polling mode).

- This callback service is in general re-entrant for multiple CAN transceiver usage, but not for the same CAN transceiver.⌋()

[CANIF823] ⌈Configuration of `<User_ConfirmPnAvailability>()`: The upper layer module, which is called (see CANIF753), has to be configurable by `CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_UL` (see CANIF820_Conf) if `CANIF_PUBLIC_PN_SUPPORT` (see CANIF772_Conf) equals True.⌋()

[CANIF824] ⌈Configuration of `<User_ConfirmPnAvailability>()`: The name of `<User_ConfirmPnAvailability>()` shall be configurable by `CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_NAME` (see CANIF819_Conf) if `CANIF_PUBLIC_PN_SUPPORT` (see CANIF772_Conf) equals True.⌋()

[CANIF825] ⌈Configuration of `<User_ConfirmPnAvailability>()`: It shall be configurable by `CANIF_PUBLIC_PN_SUPPORT` (see CANIF772_Conf), if CanIf supports this service (False: not supported, True: supported),

[CANIF826] ⌈Configuration of `<User_ConfirmPnAvailability>()`: If `CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_UL` is set to `CAN_SM`, the following is prescribed:

- `CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_NAME` must be `CanSM_ConfirmPnAvailability`
- function parameter of type `uint8` has to be named as `TransceiverId`⌋()

[CANIF827] ⌈Configuration of `<User_ConfirmPnAvailability>()`: If `CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_UL` is set to `CDD`, the following is prescribed:

- name of the service has to be configurable via parameter `CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_NAME`
- function parameter has to be of type `uint8`⌋()

8.6.3.6 <User_ClearTrcvWufFlagIndication>

[CANIF788] ⌈

Service name:	<User_ClearTrcvWufFlagIndication>	

Syntax:	`void <User_ClearTrcvWufFlagIndication>(`
	`    uint8 TransceiverId`
	`)`	

Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	TransceiverId	Abstracted CanIf TransceiverId, for which this function was called.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	

Description:	This service indicates that the CAN transceiver has cleared the WufFlag. This function is called in CanIf_ClearTrcvWufFlagIndication.

⌋()

Note: This callback service is mainly implemented in and used by the CanSm (see Specification of CAN State Manager [11]).

[CANIF793] ⌈Caveats of <`User_ClearTrcvWufFlagIndication>()`:

- The CanTrcvDrv must be initialized after Power ON.
- The call context is either on interrupt level (interrupt mode) or on task level (polling mode).
- This callback service is in general re-entrant for multiple CAN transceiver usage, but not for the same CAN transceiver.⌋()

[CANIF794] ⌈Configuration of `<User_ClearTrcvWufFlagIndication>()`: The upper layer module, which is called (see CANIF757), has to be configurable by `CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_UL` (see CANIF790_Conf) if `CANIF_PUBLIC_PN_SUPPORT` (see CANIF772_Conf) equals True.⌋()

[CANIF795] ⌈Configuration of `<User_ClearTrcvWufFlagIndication>()`: The name of `<User_ClearTrcvWufFlagIndication>()` shall be configurable by `CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_NAME` (see CANIF789_Conf) if `CANIF_PUBLIC_PN_SUPPORT` (see CANIF772_Conf) equals True.⌋()

[CANIF796] ⌈Configuration of `<User_ClearTrcvWufFlagIndication>()`: It shall be configurable by `CANIF_PUBLIC_PN_SUPPORT` (see CANIF772_Conf), if CanIf supports this service (False: not supported, True: supported),

[CANIF797] ⌈Configuration of `<User_ClearTrcvWufFlagIndication>()`: If `CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_UL` is set to `CAN_SM`, the following is prescribed:

- `CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_NAME` must be `CanSM_ClearTrcvWufFlagIndication`
- function parameter of type `uint8` has to be named as `TransceiverId`⌋()

[CANIF798] ⌈Configuration of `<User_ClearTrcvWufFlagIndication>()`: If `CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_UL` is set to `CDD`, the following is prescribed:

- name of the service has to be configurable via parameter `CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_NAME`
- function parameter has to be of type `uint8`⌋`()`

8.6.3.7 <User_CheckTrcvWakeFlagIndication>

[CANIF814] ⌈

Service name:	<User_CheckTrcvWakeFlagIndication>	

Syntax:	`void <User_CheckTrcvWakeFlagIndication>(`
	`    uint8 TransceiverId`
	`)`	

Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	TransceiverId	Abstracted CanIf TransceiverId, for which this function was called.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service indicates that the wake up flag in the CAN transceiver is set. This function is called in CanIf_CheckTrcvWakeFlagIndication.	

⌋()

Note: This callback service is mainly implemented in and used by the CanSm (see Specification of CAN State Manager [11]).

[CANIF799] ⌈Caveats of `<User_CheckTrcvWakeFlagIndication>()`:

- The CanTrcvDrv must be initialized after Power ON.
- The call context is either on interrupt level (interrupt mode) or on task level (polling mode).
- This callback service is in general re-entrant for multiple CAN transceiver usage, but not for the same CAN transceiver.⌋()

[CANIF800] ⌈Configuration of `<User_CheckTrcvWakeFlagIndication>()`: The upper layer module, which is called (see CANIF759), has to be configurable by `CANIF_DISPATCH_USERCHECKRCVWAKEFLAGINDICATION_UL` (see CANIF792_Conf) if `CANIF_PUBLIC_PN_SUPPORT` (see CANIF772_Conf) equals True.⌋()

[CANIF801] ⌈Configuration of `<User_CheckTrcvWakeFlagIndication>()`: The name of `<User_CheckTrcvWakeFlagIndication>()` shall be configurable by `CANIF_DISPATCH_USERCHECKRCVWAKEFLAGINDICATION_NAME` (see CANIF791_Conf) if `CANIF_PUBLIC_PN_SUPPORT` (see CANIF772_Conf) equals True.⌋()

[CANIF802] ⌈Configuration of `<User_CheckTrcvWakeFlagIndication>()`: It shall be configurable by `CANIF_PUBLIC_PN_SUPPORT` (see CANIF772_Conf), if CanIf supports this service (False: not supported, True: supported),

[CANIF803] ⌈Configuration of <User_CheckTrcvWakeFlagIndication>(): If CANIF_DISPATCH_USERCHECKRCVWAKEFLAGINDICATION_UL is set to CAN_SM, the following is prescribed:

- CANIF_DISPATCH_USERCHECKRCVWAKEFLAGINDICATION_NAME must be CanSM_CheckTrcvWakeFlagIndication
- function parameter of type uint8 has to be named as TransceiverId⌋()

[CANIF804] ⌈Configuration of <User_CheckTrcvWakeFlagIndication>(): If CANIF_DISPATCH_USERCHECKRCVWAKEFLAGINDICATION_UL is set to CDD, the following is prescribed:

- name of the service has to be configurable via parameter CANIF_DISPATCH_USERCHECKRCVWAKEFLAGINDICATION_NAME
- function parameter has to be of type uint8⌋()

8.6.3.8 <User_ControllerModeIndication>

[CANIF687] ⌈

Service name:	<User_ControllerModeIndication>	

Syntax:	`void <User_ControllerModeIndication>(`
	`    uint8 ControllerId,`
	`    CanIf_ControllerModeType ControllerMode`
	`)`	

Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	ControllerId	Abstracted CanIf ControllerId which is assigned to a CAN controller, at which a controller state transition occurred.
	ControllerMode	Notified CAN controller mode
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service indicates a CAN controller state transition to the corresponding upper layer module (mainly the CAN State Manager module).	

⌋()

Note: The upper layer module provides the Service ID.

Note: This callback service is mainly implemented in and used by the CAN State Manager module (see Specification of CAN State Manager [11]).

Note: The CanIf calls this callback service. It is implemented by the configured upper layer module. It is called in case of a state transition notification via CanIf_ControllerModeIndication() of the CanDrv. The delivered parameter ControllerId of the service CanIf_ControllerModeIndication() is passed to the upper layer module. The delivered parameter ControllerMode of the service CanIf_ControllerModeIndication() is mapped to the appropriate parameter ControllerMode of <User_ControllerModeIndication>().

Note: For different upper layer users different service names shall be used.

[CANIF688] ⌈Caveats of `<User_ControllerModeIndication>()`:

- The CanDrv must be initialized after Power ON.
- The call context is either on task level (polling mode).
- This callback service is in general re-entrant for multiple CAN controller usage, but not for the same CAN controller.⌋()

[CANIF689] ⌈Configuration of `<User_ControllerModeIndication>()`: The upper layer module which provides this callback service has to be configured by `CANIF_USERCONTROLLERMODEINDICATION_UL` (see CANIF684_Conf). ⌋()

[CANIF690] ⌈Configuration of `<User_ControllerModeIndication>()`: The name of `<User_ControllerModeIndication>()` which is called by the CanIf shall be configured for the CanIf by parameter `CANIF_DISPATCH_USERCTRLMODEINDICATION_NAME` (see CANIF683_Conf). This is only necessary if state transition notifications are configured via `CANIF_DISPATCH_USERCTRLMODEINDICATION_UL.`⌋()

[CANIF691] ⌈Configuration of `<User_ControllerModeIndication>()`: If `CANIF_DISPATCH_USERCTRLMODEINDICATION_UL` is set to `CAN_SM`, the following is prescribed:

- `CANIF_DISPATCH_USERCTRLMODEINDICATION_NAME` must be `CanSM_ControllerModeIndication`
- function parameter of type `uint8` has to be named as `ControllerId`⌋()

[CANIF692] ⌈Configuration of `<User_ControllerModeIndication>()`: If `CANIF_DISPATCH_USERCTRLMODEINDICATION_UL` is set to `CDD` the name of the function has to be configured via parameter `CANIF_DISPATCH_USERCTRLMODEINDICATION_NAME.` The function parameter has to be of type `uint8.`⌋()

8.6.3.9 <User_TrcvModeIndication>

[CANIF693] ⌈

Service name:	<User_TrcvModeIndication>	

Syntax:	`void <User_TrcvModeIndication>(`
	`    uint8 TransceiverId,`
	`    CanTrcv_TrcvModeType TransceiverMode`
	`)`	

Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	TransceiverId	Abstracted CanIf TransceiverId which is assigned to a CAN transceiver, at which a transceiver state transition occurred.
	TransceiverMode	Notified CAN transceiver mode
Parameters (inout):	None	

Parameters (out):	None
Return value:	None
Description:	This service indicates a CAN transceiver state transition to the corresponding upper layer module (mainly the CAN State Manager module).

⌋()

Note: The upper layer module provides the Service ID.

Note: This callback service is mainly implemented in and used by the CAN State Manager module (see Specification of CAN State Manager [11]).

Note: The CanIf calls this callback service. It is implemented by the configured upper layer module. It is called in case of a state transition notification via `CanIf_TrcvModeIndication()` of the CanTrcv. The delivered parameter `Transceiver` of the service `CanIf_TrcvModeIndication()` is mapped (as configured) to the appropriate parameter `TransceiverId` which will be passed to the upper layer module. The delivered parameter `TransceiverMode` of the service `CanIf_TrcvModeIndication()` is mapped to the appropriate parameter `TransceiverMode` of `<User_TrcvModeIndication>()`.

Note: For different upper layer users different service names shall be used.

[CANIF694] ⌈Caveats of `<User_TrcvModeIndication>()`:

- The CanTrcv must be initialized after Power ON.
- The call context is either on task level (polling mode).
- This callback service is in general re-entrant for multiple CAN transceiver usage, but not for the same CAN transceiver.⌋()

[CANIF695] ⌈Configuration of `<User_TrcvModeIndication>()`:
The upper layer module which provides this callback service has to be configured by `CANIF_DISPATCH_USERTRCVMODEINDICATION_UL` (see CANIF686_Conf), but:

- If no upper layer modules are configured for transceiver mode indications using `<User_TrcvModeIndication>()`, no transceiver mode indication needs to be configured. `CANIF_DISPATCH_USERTRCVMODEINDICATION_UL` needs not to be configured.
- If transceivers are not supported (`CanInterfaceTransceiverDriverConfiguration` ais not configured, see CANIF273_Conf), `CANIF_DISPATCH_USERTRCVMODEINDICATION_UL` is not configurable.⌋()

If no upper layer modules are configured for state transition notifications using `<User_TrcvModeIndication>()`, no state transition notification needs to be configured.

[CANIF696] ⌈Configuration of `<User_TrcvModeIndication>()`: The name of `<User_TrcvModeIndication>()` which will be called by the CAN Interface module shall be configured for the CAN Interface module by parameter

CANIF_DISPATCH_USERTRCVMODEINDICATION_NAME (see CANIF685_Conf). This is only necessary if state transition notifications are configured via CANIF_DISPATCH_USERTRCVMODEINDICATION_UL.⌋()

[CANIF697] ⌈Configuration of <User_TrcvModeIndication>(): If CANIF_DISPATCH_USERTRCVMODEINDICATION_UL is set to CAN_SM, the following is prescribed:

- CANIF_DISPATCH_USERTRCVMODEINDICATION_NAME must be CanSM_TransceiverModeIndication
- function parameter of type uint8 has to be named as TransceiverId⌋()

[CANIF698] ⌈Configuration of <User_TrcvModeIndication>(): If CANIF_DISPATCH_USERTRCVMODEINDICATION_UL is set to CDD the name of the API has to be configured via parameter CANIF_DISPATCH_USERTRCVMODEINDICATION_NAME. The function parameter has to be of type uint8.⌋()

9 Sequence diagrams

The following sequence diagrams show the interactions between CanIf and CanDrv.

9.1 Transmit request (single CAN Driver)

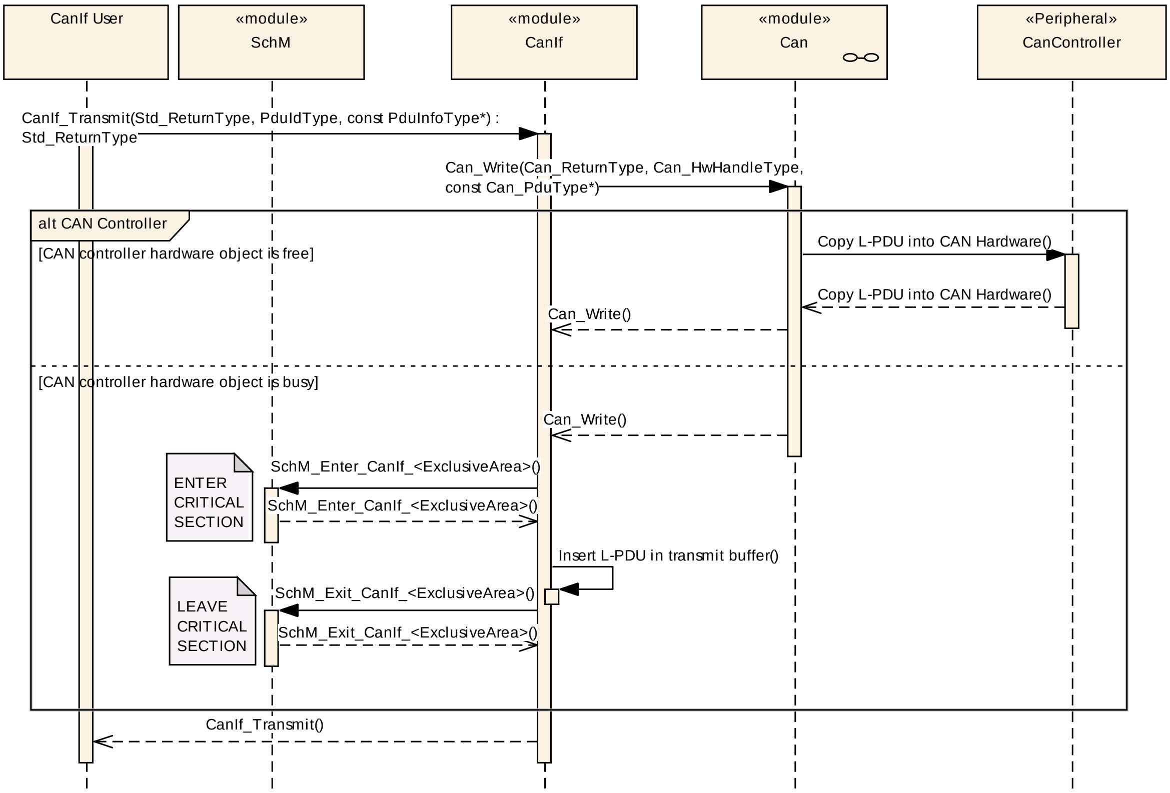

CanIf User
«module»
SchM
«module»
CanIf
«module»
Can
«Peripheral»
CanController
CanIf_Transmit(Std_ReturnType, PduIdType, const PduInfoType*) :
Std_ReturnType
Can_Write(Can_ReturnType, Can_HwHandleType,
const Can_PduType*)
alt CAN Controller
[CAN controller hardware object is free]
Copy L-PDU into CAN Hardware()
Copy L-PDU into CAN Hardware()
Can_Write()
[CAN controller hardware object is busy]
Can_Write()
SchM_Enter_CanIf_<ExclusiveArea>()
ENTER
CRITICAL
SECTION
SchM_Enter_CanIf_<ExclusiveArea>()
Insert L-PDU in transmit buffer()
SchM_Exit_CanIf_<ExclusiveArea>()
LEAVE
CRITICAL
SECTION
SchM_Exit_CanIf_<ExclusiveArea>()
CanIf_Transmit()

Figure 19 Transmission request with a single CAN Driver

Activity	***Description***
Transmission request	The upper layer initiates a transmit request via the service `CanIf_Transmit()`. The parameter CanTxPduId identifies the requested L-PDU. The service performs following steps: - validation of the input parameter - definition of the CAN controller to be used The second parameter *PduInfoPtr is a pointer on the structure with transmit L-PDU related data such as CanSduLength and *CanSduPtr.
Start transmission	`CanIf_Transmit()` requests a transmission and calls the CanDrv service `Can_Write()` with corresponding processing of the HTH.
Hardware request	`Can_Write()`writes all L-PDU data in the CAN Hardware (if it is free) and sets the hardware request for transmission.
E_OK from Can_Write service	`Can_Write()` returns E_OK to `CanIf_Transmit().`
E_BUSY from Can_Write service	If the CanDrv detects, there are no free hardware objects available, it returns `CAN_E_BUSY` to the CanIf.
Copying into the buffer	The L-PDU of the rejected transmit request will be inserted in the transmit buffer of the CanIf until the next transmit confirmation.

E_OK from CAN Interface	`CanIf_Transmit()` returns E_OK to the upper layer.

9.2 Transmit request (multiple CAN Drivers)

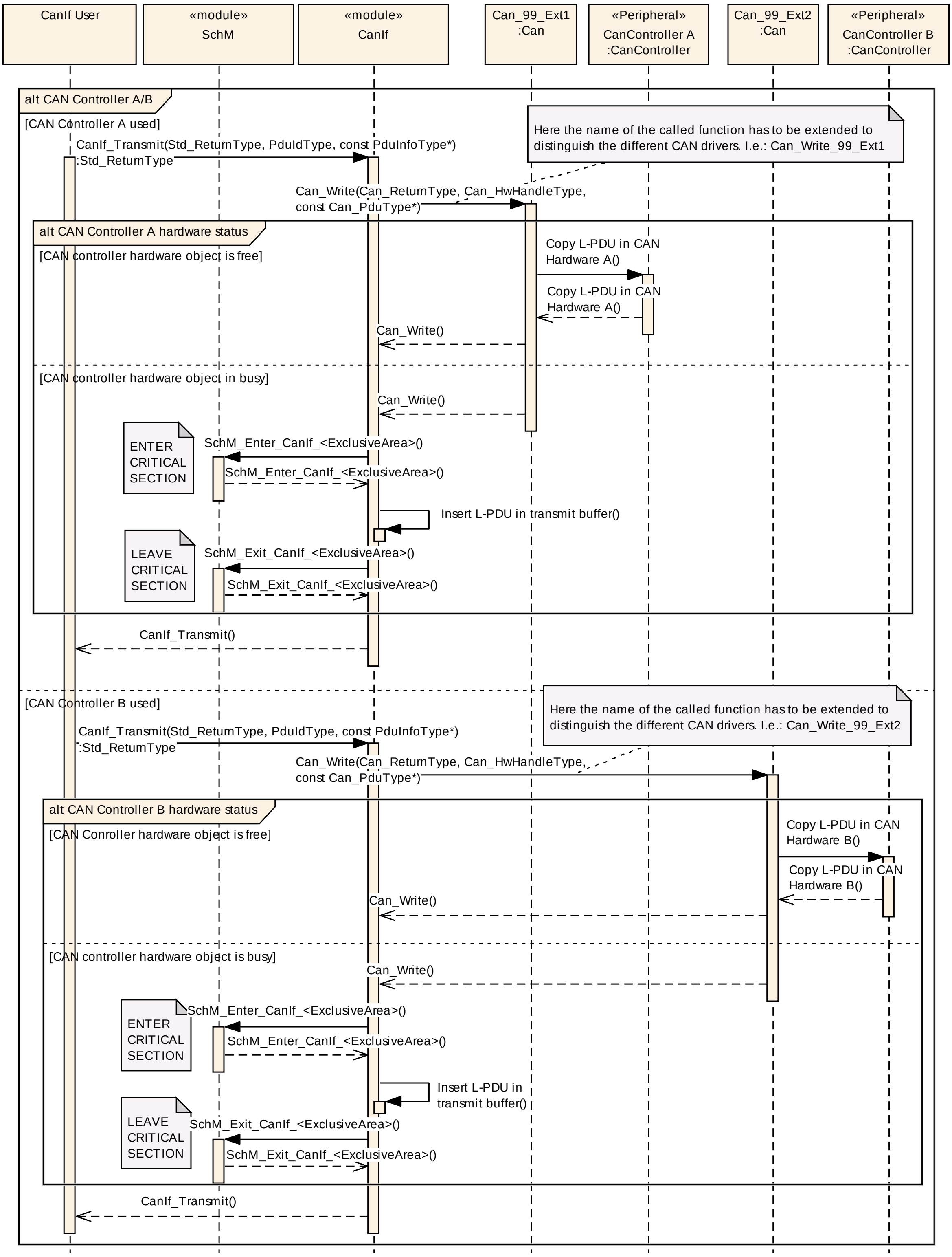

CanIf User
«module» SchM
«module» CanIf
Can_99_Ext1 :Can
«Peripheral» CanController A :CanController
Can_99_Ext2 :Can
«Peripheral» CanController B :CanController
alt CAN Controller A/B
[CAN Controller A used]
CanIf_Transmit(Std_ReturnType, PduIdType, const PduInfoType*) :Std_ReturnType
Here the name of the called function has to be extended to distinguish the different CAN drivers. I.e.: Can_Write_99_Ext1
Can_Write(Can_ReturnType, Can_HwHandleType, const Can_PduType*)
alt CAN Controller A hardware status
[CAN controller hardware object is free]
Copy L-PDU in CAN Hardware A()
Copy L-PDU in CAN Hardware A()
Can_Write()
[CAN controller hardware object in busy]
Can_Write()
ENTER CRITICAL SECTION
SchM_Enter_CanIf_<ExclusiveArea>()
SchM_Enter_CanIf_<ExclusiveArea>()
Insert L-PDU in transmit buffer()
LEAVE CRITICAL SECTION
SchM_Exit_CanIf_<ExclusiveArea>()
SchM_Exit_CanIf_<ExclusiveArea>()
CanIf_Transmit()
[CAN Controller B used]
Here the name of the called function has to be extended to distinguish the different CAN drivers. I.e.: Can_Write_99_Ext2
CanIf_Transmit(Std_ReturnType, PduIdType, const PduInfoType*) :Std_ReturnType
Can_Write(Can_ReturnType, Can_HwHandleType, const Can_PduType*)
alt CAN Controller B hardware status
[CAN Conroller hardware object is free]
Copy L-PDU in CAN Hardware B()
Copy L-PDU in CAN Hardware B()
Can_Write()
[CAN controller hardware object is busy]
Can_Write()
ENTER CRITICAL SECTION
SchM_Enter_CanIf_<ExclusiveArea>()
SchM_Enter_CanIf_<ExclusiveArea>()
Insert L-PDU in transmit buffer()
LEAVE CRITICAL SECTION
SchM_Exit_CanIf_<ExclusiveArea>()
SchM_Exit_CanIf_<ExclusiveArea>()
CanIf_Transmit()

Figure 20 Transmission request with multiple CAN Drivers

First transmit request:

Activity	*Description*
Transmission request A	The upper layer initiates a transmit request via the service `CanIf_Transmit()`. The parameter CanTxPduId identifies the requested L-PDU. The service performs following steps: - validation of the input parameter - definition of the CAN controller to be used (here: Can_99_Ext1) The second parameter *PduInfoPtr is a pointer on the structure with transmit L-PDU related data such as CanSduLength and *CanSduPtr.
Start transmission	`CanIf_Transmit()` requests a transmission and calls the CanDrv Can_99_Ext1 service `Can_Write_99_Ext1()` with corresponding processing of the HTH.
Hardware request	`Can_Write_99_Ext1()`writes all L-PDU data in the CAN Hardware of Controller A (if it is free) and sets the hardware request for transmission.
E_OK from Can_Write service	`Can_Write_99_Ext1()` returns E_OK to `CanIf_Transmit().`
E_BUSY from Can_Write service	If the CanDrv Can_99_Ext1 detects, there are no free hardware objects available, it returns `CAN_E_BUSY` to the CanIf.
Copying into the buffer	The L-PDU of the rejected transmit request will be inserted in the transmit buffers of the CAN Interface until the next transmit confirmation.
E_OK from CAN Interface	`CanIf_Transmit()` returns E_OK to the upper layer.

Second transmit request:

Activity	*Description*
Transmission request B	The upper layer initiates a transmit request via the service `CanIf_Transmit()`. The parameter CanTxPduId identifies the requested L-PDU. The service performs following steps: - validation of the input parameter - definition of the CAN controller to be used (here: Can_99_Ext2) The second parameter *PduInfoPtr is a pointer on the structure with receive L-PDU related data such as CanSduLength and *CanSduPtr.
Start transmission	`CanIf_Transmit()` starts a transmission and calls the CanDrv Can_99_Ext2 service `Can_Write_99_Ext2()` with corresponding processing of the HTH.
Hardware request	`Can_Write_99_Ext2 ()`writes all L-PDU data in the CAN Hardware of Controller B (if it is free) and sets the hardware request for transmission.
E_OK from Can_Write service	`Can_Write_99_Ext2 ()` returns E_OK to `CanIf_Transmit().`
E_BUSY from Can_Write service	If the CAN Driver module Can_99_`Ext2` detects, there are no free hardware objects available, it returns `CAN_E_BUSY` to the CAN Interface.
Copying into the buffer	The L-PDU of the rejected transmit request will be inserted in the transmit buffers of the CAN Interface until the next transmit confirmation.
E_OK from CAN Interface	`CanIf_Transmit()` returns E_OK to the upper layer.

9.3 Transmit confirmation (interrupt mode)

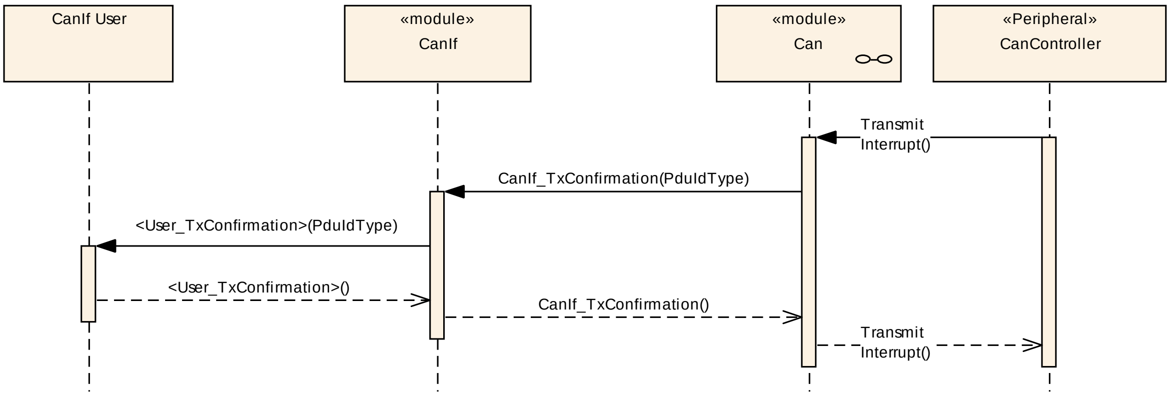


Figure 21 Transmit confirmation interrupt driven

Activity	***Description***
Transmit interrupt	The acknowledged CAN frame signals a successful transmission to the receiving CAN controller and triggers the transmit interrupt.
Confirmation to the CAN Interface	CAN Driver calls the service `CanIf_TxConfirmation()`. The parameter CanTxPduId specifies the CAN L-PDU previously sent by `Can_Write()`. The CAN diver must store the all in HTHs pending L-PDU Ids in an array organized per HTH to avoid new search of the L-PDU ID for call of `CanIf_TxConfirmation()`.
Confirmation to upper layer	Calling of the corresponding upper layer confirmation service `<User_TxConfirmation>()`. It signals a successful L-PDU transmission to the upper layer.

9.4 Transmit confirmation (polling mode)

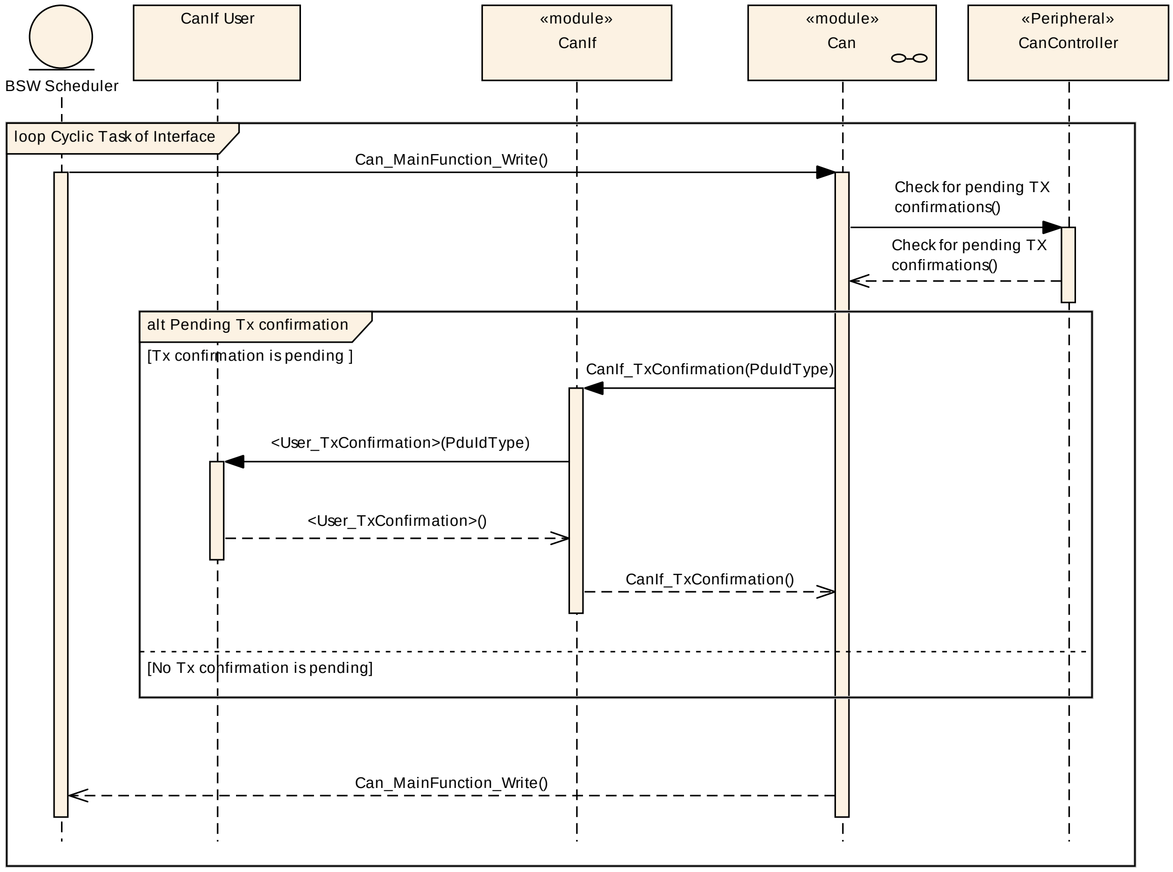

BSW Scheduler
CanIf User
«module»
CanIf
«module»
Can
«Peripheral»
CanController
loop Cyclic Task of Interface
Can_MainFunction_Write()
Check for pending TX confirmations()
Check for pending TX confirmations()
alt Pending Tx confirmation
[Tx confirmation is pending]
CanIf_TxConfirmation(PduIdType)
<User_TxConfirmation>(PduIdType)
<User_TxConfirmation>()
CanIf_TxConfirmation()
[No Tx confirmation is pending]
Can_MainFunction_Write()

Figure 22 Transmit confirmation polling driven

Activity	***Description***
Cyclic Task **CAN Driver**	The service `Can_MainFunction_Write()`is called by the BSW Scheduler.
Check for pending transmit confirmations	`Can_MainFunction_Write()`checks the underlying CAN controller(s) about pending transmit confirmations of previously succeeded transmit events.
Transmit Confirmation	The acknowledged CAN frame signals a successful transmission to the sending CAN controller.
Confirmation to CAN Interface	CAN Driver calls the service `CanIf_TxConfirmation()` The parameter CanTxPduId specifies the CAN L-PDU previously sent by `Can_Write()`. The CAN diver must store the all in HTHs pending L-PDU Ids in an array organized per HTH to avoid new search of the L-PDU ID for call of `CanIf_TxConfirmation()`.
Confirmation to upper layer	Calling of the corresponding upper layer confirmation service `<User_TxConfirmation>()`. It signals a successful L-PDU transmission to the upper layer.

9.5 Transmit confirmation (with buffering)

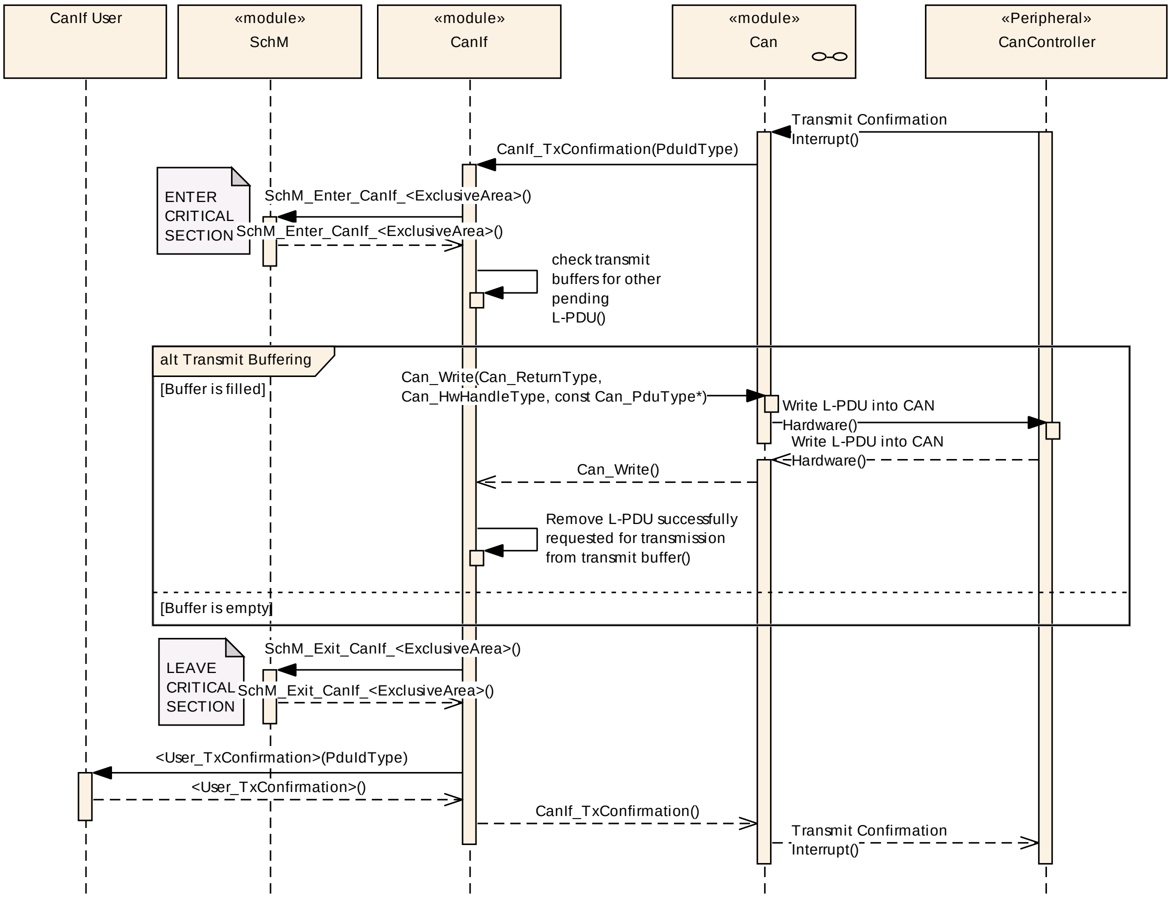

CanIf User
«module»
SchM
«module»
CanIf
«module»
Can
«Peripheral»
CanController
Transmit Confirmation Interrupt()
CanIf_TxConfirmation(PduIdType)
ENTER CRITICAL SECTION
SchM_Enter_CanIf_<ExclusiveArea>()
SchM_Enter_CanIf_<ExclusiveArea>()
check transmit buffers for other pending L-PDU()
alt Transmit Buffering
[Buffer is filled]
Can_Write(Can_ReturnType, Can_HwHandleType, const Can_PduType*)
Write L-PDU into CAN Hardware()
Write L-PDU into CAN Hardware()
Can_Write()
Remove L-PDU successfully requested for transmission from transmit buffer()
[Buffer is empty]
LEAVE CRITICAL SECTION
SchM_Exit_CanIf_<ExclusiveArea>()
SchM_Exit_CanIf_<ExclusiveArea>()
<User_TxConfirmation>(PduIdType)
<User_TxConfirmation>()
CanIf_TxConfirmation()
Transmit Confirmation Interrupt()

Figure 23 Transmit confirmation with buffering

Activity	***Description***
Transmit interrupt	Acknowledged CAN frame signals successful transmission to receiving CAN controller and triggers transmit interrupt.
Confirmation to CAN Interface	CanDrv calls service `CanIf_TxConfirmation()`. Parameter CanTxPduId specifies the CAN L-PDU previously transmitted by `Can_Write()`. CanDrv must store the L-PDU IDs of all in HTHs pending L-PDUs in an array organized per HTH to avoid new search of the L-PDU ID for call of `CanIf_TxConfirmation()`.
ENTER CRITICAL SECTION	Protect transmit buffers from being corrupted. This is done by entering an exclusive area defined in the SchM.
Check of transmit buffers	The transmit buffers of the CanIf checked, whether a pending L-PDU is stored or not.
Transmit request passed to the CAN Driver	In case of pending L-PDUs in the transmit buffers the highest priority order the latest L-PDU is requested for transmission by `Can_Write()`. It signals a successful L-PDU transmission to the upper layer. Thus `Can_Write()` can be called re-entrant.
Remove transmitted L-PDU from transmit buffers	The L-PDU pending for transmission is removed from the transmission buffers by the CanIf.
LEAVE CRITICAL SECTION	End of protection segment.
Confirmation to the upper layer	Calling of the corresponding upper layer confirmation service `<User_TxConfirmation>()`. It signals a

	successful L-PDU transmission to the upper layer.

9.6 Transmit cancellation (with buffering)

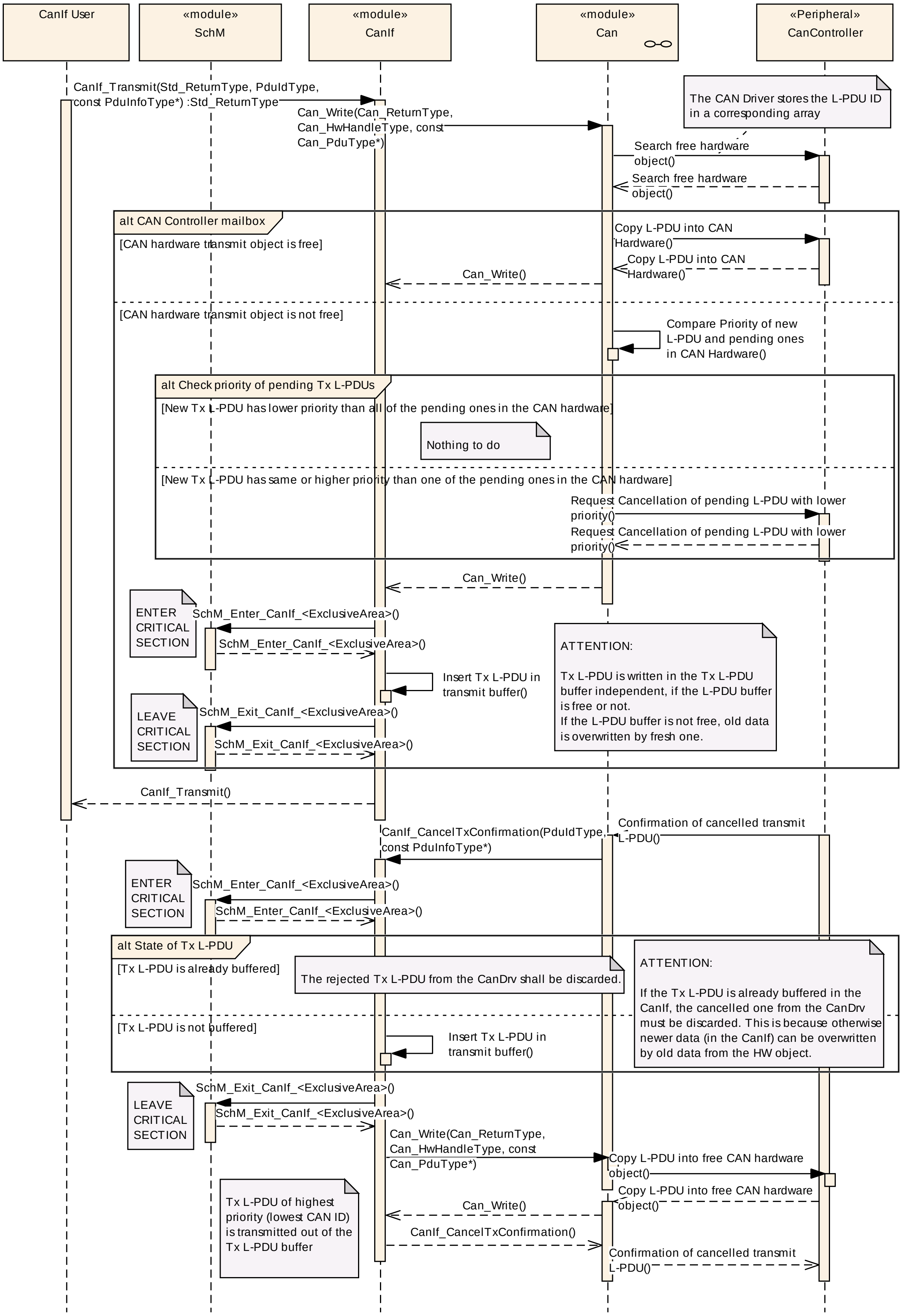

CanIf User
«module» SchM
«module» CanIf
«module» Can
«Peripheral» CanController
CanIf_Transmit(Std_ReturnType, PduIdType, const PduInfoType*) :Std_ReturnType
Can_Write(Can_ReturnType, Can_HwHandleType, const Can_PduType*)
The CAN Driver stores the L-PDU ID in a corresponding array
Search free hardware object()
Search free hardware object()
alt CAN Controller mailbox
[CAN hardware transmit object is free]
Copy L-PDU into CAN Hardware()
Copy L-PDU into CAN Hardware()
Can_Write()
[CAN hardware transmit object is not free]
Compare Priority of new L-PDU and pending ones in CAN Hardware()
alt Check priority of pending Tx L-PDUs
[New Tx L-PDU has lower priority than all of the pending ones in the CAN hardware]
Nothing to do
[New Tx L-PDU has same or higher priority than one of the pending ones in the CAN hardware]
Request Cancellation of pending L-PDU with lower priority()
Request Cancellation of pending L-PDU with lower priority()
Can_Write()
ENTER CRITICAL SECTION
SchM_Enter_CanIf_<ExclusiveArea>()
SchM_Enter_CanIf_<ExclusiveArea>()
Insert Tx L-PDU in transmit buffer()
ATTENTION:
Tx L-PDU is written in the Tx L-PDU buffer independent, if the L-PDU buffer is free or not.
If the L-PDU buffer is not free, old data is overwritten by fresh one.
LEAVE CRITICAL SECTION
SchM_Exit_CanIf_<ExclusiveArea>()
SchM_Exit_CanIf_<ExclusiveArea>()
CanIf_Transmit()
CanIf_CancelTxConfirmation(PduIdType, const PduInfoType*)
Confirmation of cancelled transmit L-PDU()
ENTER CRITICAL SECTION
SchM_Enter_CanIf_<ExclusiveArea>()
SchM_Enter_CanIf_<ExclusiveArea>()
alt State of Tx L-PDU
[Tx L-PDU is already buffered]
The rejected Tx L-PDU from the CanDrv shall be discarded.
ATTENTION:
If the Tx L-PDU is already buffered in the CanIf, the cancelled one from the CanDrv must be discarded. This is because otherwise newer data (in the CanIf) can be overwritten by old data from the HW object.
[Tx L-PDU is not buffered]
Insert Tx L-PDU in transmit buffer()
LEAVE CRITICAL SECTION
SchM_Exit_CanIf_<ExclusiveArea>()
SchM_Exit_CanIf_<ExclusiveArea>()
Can_Write(Can_ReturnType, Can_HwHandleType, const Can_PduType*)
Copy L-PDU into free CAN hardware object()
Copy L-PDU into free CAN hardware object()
Can_Write()
Tx L-PDU of highest priority (lowest CAN ID) is transmitted out of the Tx L-PDU buffer
CanIf_CancelTxConfirmation()
Confirmation of cancelled transmit L-PDU()

Figure 24 Transmit cancellation (with buffering)

Activity	***Description***
Transmission request	The upper layer initiates a transmit request via the service `CanIf_Transmit()`. The parameter CanTxPduId identifies the requested L-PDU. The service performs following steps: - validation of the input parameter - definition of the CAN controller to be used The second parameter *PduInfoPtr is a pointer on the structure with transmit L-PDU related data such as CanSduLength and *CanSduPtr.
Start transmission	`CanIf_Transmit()` requests a transmission and calls the CanDrv service `Can_Write()` with corresponding processing of the HTH.
Hardware request	`Can_Write()`writes all L-PDU data in the CAN Hardware (if it is free) and sets the hardware request for transmission.
E_OK from Can_Write service	`Can_Write()` returns E_OK to `CanIf_Transmit().`
E_BUSY from Can_Write service without transmit abort	If the CanDrv detects, there are no free hardware objects available and the new transmit L-PDU has lower priority than all of the pending ones in the CAN hardware have, it returns `CAN_E_BUSY` to the CanIf.
E_BUSY from Can_Write service with transmit abort	If the CanDrv detects, there are no free hardware objects available and the new transmit L-PDU has higher priority than all of the pending ones in the CAN hardware, it requested transmit abort of the pending L-PDU in the CAN hardware with the lowest priority and returns `CAN_E_BUSY` to the CanIf.
Transmit buffer	The CanIf stores the rejected L-PDU in the transmit buffers.
E_OK from CAN Interface	`CanIf_Transmit()` returns E_OK to the upper layer.

Cancellation confirmation notification:

Activity	***Description***
Transmit cancellation confirmation interrupt	CAN controller signals a successful aborted CAN L-PDU. CanDrv detects the abort confirmation event either by interrupt or polling.
Confirmation to CAN Interface	CanDrv calls service `CanIf_CancelTxConfirmation()`. The parameter CanTxPduId specifies the CAN L-PDU successfully aborted by the CanDrv. The CanDrv must store the all in HTHs pending L-PDU Ids in an array organized per HTH to avoid new search of the L-PDU ID for call of `CanIf_CancelTxConfirmation()`.
ENTER CRITICAL SECTION	Protect transmit buffers from being corrupted. This is done by entering an exclusive area defined in the SchM.
Check of transmit buffers	The transmit buffer of the CanIf checked, whether the L-PDU with the same CanTxPduId is already stored or not. If yes, the cancelled L-PDU is lost. If not, the cancelled L-PDU is stored in the transmit buffer.
Transmit request passed to the CAN Driver	Pending L-PDUs in the transmit buffers with the highest priority order is requested for transmission by `Can_Write()`. It signals a successful L-PDU transmission to the upper layer. Thus `Can_Write()` calls can occur re-entrant.
Remove transmited L-PDU from transmit buffers	The L-PDU pending for transmission is removed from the transmission buffers by the CanIf.
LEAVE CRITICAL SECTION	End of protection segment.
Cancellation confirmation finished	The cancellation confirmation callback returns.

9.7 Transmit cancellation

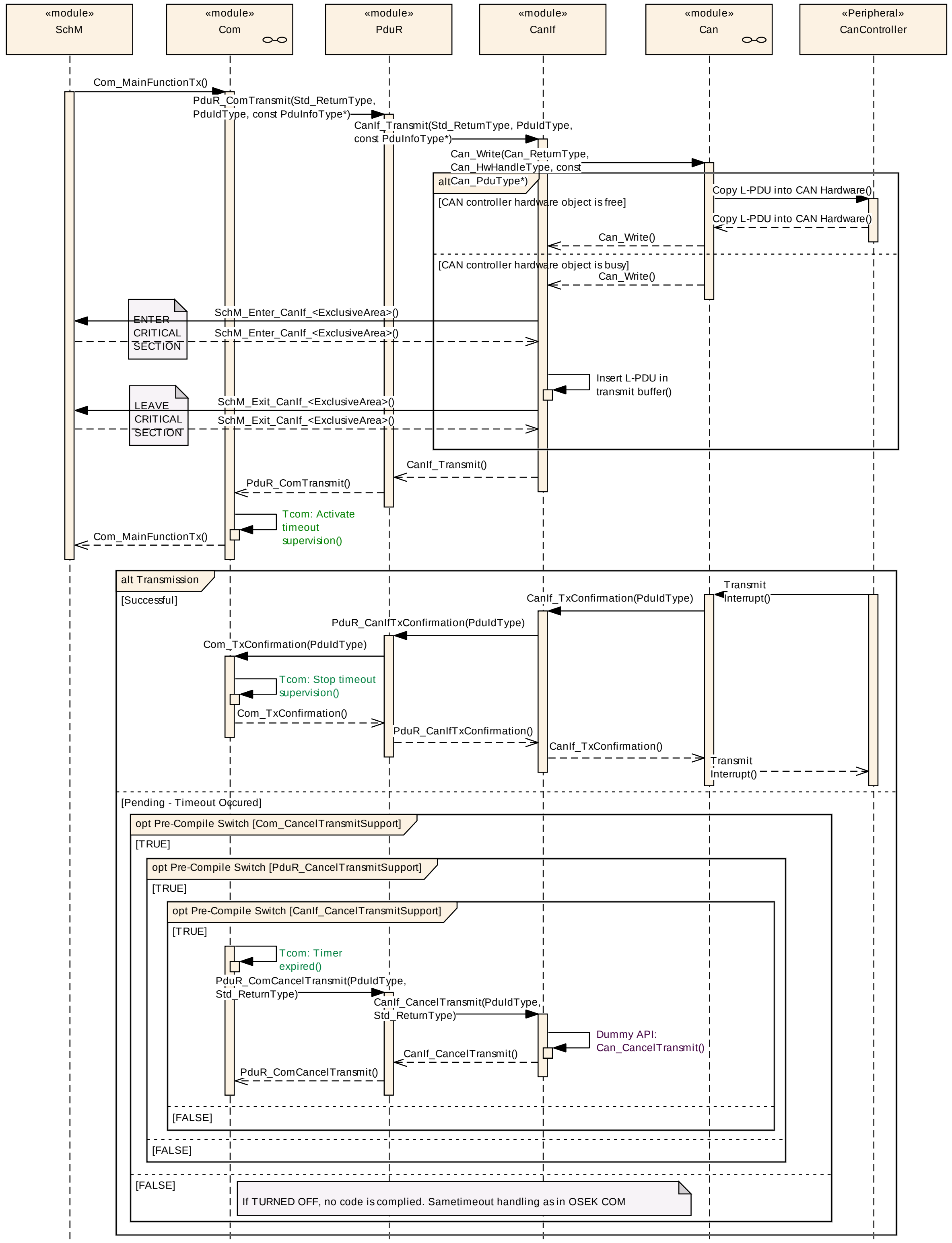

«module»
SchM
«module»
Com
«module»
PduR
«module»
CanIf
«module»
Can
«Peripheral»
CanController
Com_MainFunctionTx()
PduR_ComTransmit(Std_ReturnType, PduIdType, const PduInfoType*)
CanIf_Transmit(Std_ReturnType, PduIdType, const PduInfoType*)
Can_Write(Can_ReturnType, Can_HwHandleType, const Can_PduType*)
alt
[CAN controller hardware object is free]
Copy L-PDU into CAN Hardware()
Copy L-PDU into CAN Hardware()
Can_Write()
[CAN controller hardware object is busy]
Can_Write()
ENTER CRITICAL SECTION
SchM_Enter_CanIf_<ExclusiveArea>()
SchM_Enter_CanIf_<ExclusiveArea>()
Insert L-PDU in transmit buffer()
LEAVE CRITICAL SECTION
SchM_Exit_CanIf_<ExclusiveArea>()
SchM_Exit_CanIf_<ExclusiveArea>()
CanIf_Transmit()
PduR_ComTransmit()
Tcom: Activate timeout supervision()
Com_MainFunctionTx()
alt Transmission
[Successful]
Transmit Interrupt()
CanIf_TxConfirmation(PduIdType)
PduR_CanIfTxConfirmation(PduIdType)
Com_TxConfirmation(PduIdType)
Tcom: Stop timeout supervision()
Com_TxConfirmation()
PduR_CanIfTxConfirmation()
CanIf_TxConfirmation()
Transmit Interrupt()
[Pending - Timeout Occured]
opt Pre-Compile Switch [Com_CancelTransmitSupport]
[TRUE]
opt Pre-Compile Switch [PduR_CancelTransmitSupport]
[TRUE]
opt Pre-Compile Switch [CanIf_CancelTransmitSupport]
[TRUE]
Tcom: Timer expired()
PduR_ComCancelTransmit(PduIdType, Std_ReturnType)
CanIf_CancelTransmit(PduIdType, Std_ReturnType)
Dummy API: Can_CancelTransmit()
CanIf_CancelTransmit()
PduR_ComCancelTransmit()
[FALSE]
[FALSE]
[FALSE]
If TURNED OFF, no code is complied. Sametimeout handling as in OSEK COM

Figure 25 Transmit cancellation

Activity	***Description***
Call of scheduled Function	`Com_MainFunctionTx()` will be called cyclic by the SchM.
Transmission request to the PDU Router	Within cyclic called `Com_MainFunctionTx()` a transmission request through the PduR arises: `PduR_ComTransmit()`

Transmission request to the CAN Interface	PduR passes the transmit request via `CanIf_Transmit()` to the CanIf. The parameter `CanTxPduId` identifies the requested L-PDU. The service performs following steps: - validation of the input parameter - definition of the CAN controller to be used The second parameter `*PduInfoPtr` is a pointer on the structure with transmit L-PDU related data such as CanSduLength and *CanSduPtr.
Transmission request to the CAN Driver	`CanIf_Transmit()` requests a transmission and calls the CanDrv service `Can_Write()` with corresponding processing of the HTH.
Transmission request to the hardware	`Can_Write()`writes all L-PDU data in the CAN Hardware (if it is free) and sets the hardware request for transmission.
E_OK from Can_Write service	`Can_Write()` returns E_OK to `CanIf_Transmit().`
E_BUSY from Can_Write service	If the CanDrv detects, there are no free hardware objects available, it returns CAN_E_BUSY to the CanIf.
Copying into the buffer	The L-PDU of the rejected transmit request will be inserted in the transmit buffer of CanIf until the next transmit confirmation.
E_OK from CAN Interface	`CanIf_Transmit()` returns E_OK to the PduR.
E_OK from PDU Router	`PduR_ComTransmit()` returns E_OK to the COM.
Starting Timeout supervision	The PduR starts a timeout supervision which checks if a confirmation for the successful transmission will arrive.
E_OK from COM	The `Com_MainFunctionTx()` returns `E_OK` to the SchM.

Transmit confirmation interrupt driven:

Activity	***Description***
Transmit interrupt	If it appears, the acknowledged CAN frame signals a successful transmission to the receiving CAN controller and triggers the transmit interrupt.
Confirmation to the CAN Interface	CanDrv calls service `CanIf_TxConfirmation()`. Parameter `CanTxPduId` specifies the CAN L-PDU previously sent by `Can_Write()`. The CanDrv must store the all in HTHs pending L-PDU Ids in an array organized per HTH to avoid new search of the L-PDU ID for call of `CanIf_TxConfirmation()`.
Confirmation to the PDU Router	CanIf calls the service `PduR_CanIfTxConfirmation()` with the corresponding `CanTxPduId`.
Confirmation to the COM	The PDU Router informs the COM module about the successful L-PDU transmission via the API `Com_TxConfirmation()` with the corresponding `ComTxPduId`. If this happened, the timeout supervision, which has been started after the successful request for transmission has been signaled to the COM, is stopped.

Cancellation confirmation notification:

Activity	***Description***
Transmit cancellation to the PDU Router	If `Com_CancelTransmitSupport`, `PduR_CancelTransmitSupport` and `CanIf_CancelTransmitSupport` are activated, the API `PduR_ComCancelTransmit ()` is called by the COM module with the corresponding parameter `ComTxPduId` e.g. after a timer has been expired.
Transmit cancellation to the CAN Interface	If the PduR passes the transmit cancellation via the service `CanIf_CancelTransmit()` to the CanIf. The parameter `CanTxPduId` identifies the requested L-PDU.
E_NOT_OK from CanIf_CancelTransmit	The dummy function `CanIf_CancelTransmit()` returns `E_NOT_OK` to the PduR.

E_NOT_OK from PduR_ComCancelTransmit	The PduR returns E_NOT_OK to the COM module.

9.8 Receive indication (interrupt mode)

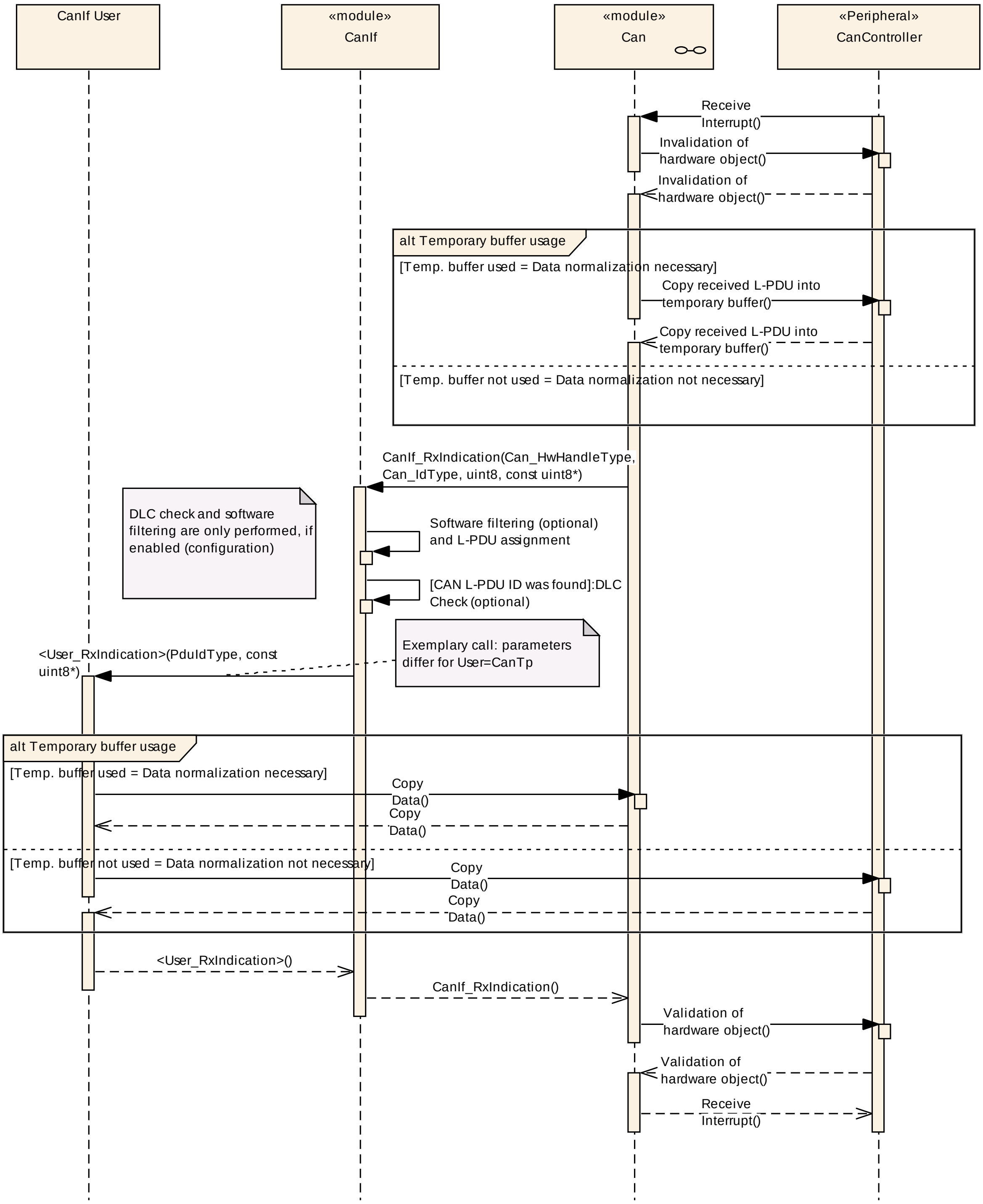

CanIf User
«module» CanIf
«module» Can
«Peripheral» CanController
Receive Interrupt()
Invalidation of hardware object()
Invalidation of hardware object()
alt Temporary buffer usage
[Temp. buffer used = Data normalization necessary]
Copy received L-PDU into temporary buffer()
Copy received L-PDU into temporary buffer()
[Temp. buffer not used = Data normalization not necessary]
CanIf_RxIndication(Can_HwHandleType, Can_IdType, uint8, const uint8*)
DLC check and software filtering are only performed, if enabled (configuration)
Software filtering (optional) and L-PDU assignment
[CAN L-PDU ID was found]:DLC Check (optional)
Exemplary call: parameters differ for User=CanTp
<User_RxIndication>(PduIdType, const uint8*)
alt Temporary buffer usage
[Temp. buffer used = Data normalization necessary]
Copy Data()
Copy Data()
[Temp. buffer not used = Data normalization not necessary]
Copy Data()
Copy Data()
<User_RxIndication>()
CanIf_RxIndication()
Validation of hardware object()
Validation of hardware object()
Receive Interrupt()

Figure 26 Receive indication interrupt driven

Activity	*Description*
Receive Interrupt	The CAN controller signals a successful reception and triggers a receive interrupt.
Invalidation of CAN hardware object,	The CPU (CAN Driver) get exclusive access rights to the

provide CPU access to CAN mailbox	CAN mailbox or at least to the corresponding hardware object, where new data were received.
Buffering, normalizing	The L-SDU is normalized and is buffered in the temporary buffer located in the CAN Driver. Each CAN Driver owns a temporary buffer for every physical channel only if normalizing of the data is necessary.
Indication to **CAN Interface**	The reception is indicated to the CAN Interface by calling of `CanIf_RxIndication()`. The HRH specifies the CAN RAM hardware object and the corresponding CAN controller, which contains the received L-PDU. The temporary buffer is referenced to the CAN Interface by *CanSduPtr.
Software Filtering	The Software Filtering checks, whether the received L-PDU will be processed on a local ECU. If not, the received L-PDU is not indicated to upper layers. Further processing is suppressed.
DLC check	If the L-PDU is found, the DLC of the received L-PDU is compared with the expected, statically configured one for the received L-PDU.
Receive Indication to the upper layer	The corresponding receive indication service of the upper layer is called. This signals a successful reception to the target upper layer. The parameter CanPduId specifies the L-PDU, the second parameter is the reference on the temporary buffer within the L-SDU. During is execution of this service the CAN hardware buffers must be unlocked for CPU access/locked for CAN controller access.
Validation of CAN hardware object, allow access of CAN controller to CAN mailbox	The CAN controller get back exclusive access rights to the CAN mailbox or at least to the corresponding hardware object, where new data were already being copied into the upper layer buffer.

9.9 Receive indication (polling mode)

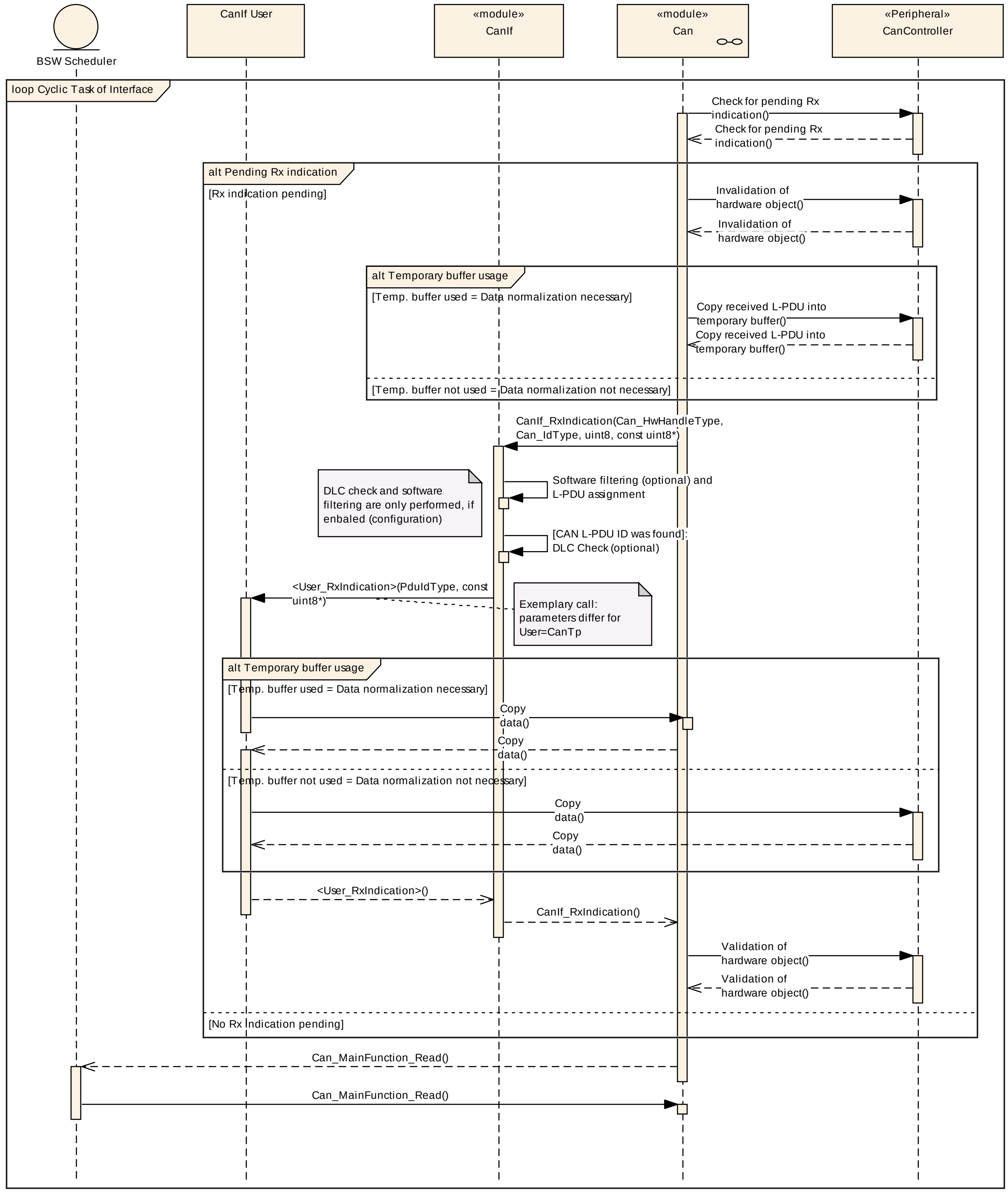

BSW Scheduler
CanIf User
«module» CanIf
«module» Can
«Peripheral» CanController
loop Cyclic Task of Interface
Check for pending Rx indication()
Check for pending Rx indication()
alt Pending Rx indication
[Rx indication pending]
Invalidation of hardware object()
Invalidation of hardware object()
alt Temporary buffer usage
[Temp. buffer used = Data normalization necessary]
Copy received L-PDU into temporary buffer()
Copy received L-PDU into temporary buffer()
[Temp. buffer not used = Data normalization not necessary]
CanIf_RxIndication(Can_HwHandleType, Can_IdType, uint8, const uint8*)
DLC check and software filtering are only performed, if enbaled (configuration)
Software filtering (optional) and L-PDU assignment
[CAN L-PDU ID was found]: DLC Check (optional)
<User_RxIndication>(PduIdType, const uint8*)
Exemplary call: parameters differ for User=CanTp
alt Temporary buffer usage
[Temp. buffer used = Data normalization necessary]
Copy data()
Copy data()
[Temp. buffer not used = Data normalization not necessary]
Copy data()
Copy data()
<User_RxIndication>()
CanIf_RxIndication()
Validation of hardware object()
Validation of hardware object()
[No Rx Indication pending]
Can_MainFunction_Read()
Can_MainFunction_Read()

Figure 27 Receive indication polling driven

Activity	*Description*
Cyclic Task **CAN Driver**	The service `Can_MainFunction_Read()`is called by the BSW Scheduler.
Check for new received L-PDU	`Can_MainFunction_Read()`checks the underlying CAN controller(s) about new received L-PDUs.
Invalidation of CAN hardware object, provide CPU access to CAN mailbox	In case of a new receive event the CPU (CAN Driver) get exclusive access rights to the CAN mailbox or at least to the corresponding hardware object, where new data were received.

Buffering, normalizing	In case of a new receive event the L-SDU is normalized and is buffered in the temporary buffer located in the CAN Driver. Each CAN Driver owns such a temporary buffer for every physical channel only if normalizing of the data is necessary.
Indication to **CAN Interface**	The reception is indicated to the CAN Interface by calling of `CanIf_RxIndication()`. The HRH specifies the CAN RAM hardware object and the corresponding CAN controller, which contains the received L-PDU. The temporary buffer is referenced to the CAN Interface by *CanSduPtr.
Software Filtering	The Software Filtering checks, whether the received L-PDU will be processed on a local ECU. If not, the received L-PDU is not indicated to upper layers. Further processing is suppressed.
DLC check	If the L-PDU is found, the DLC of the received L-PDU is compared with the expected, statically configured one for the received L-PDU.
Receive Indication to the upper layer	If configured, the corresponding receive indication service of the upper layer is called. This signals a successful reception to the target upper layer. The parameter CanPduId specifies the L-PDU, the second parameter is the reference on the temporary buffer within the L-SDU. During is execution of this service the CAN hardware buffers must be unlocked for CPU access/locked for CAN controller access.
Validation of CAN hardware object, allow access of CAN controller to CAN mailbox	The CAN controller get back exclusive access rights to the CAN mailbox or at least to the corresponding hardware object, where new data were already being copied into the upper layer buffer.

9.10 Read received data

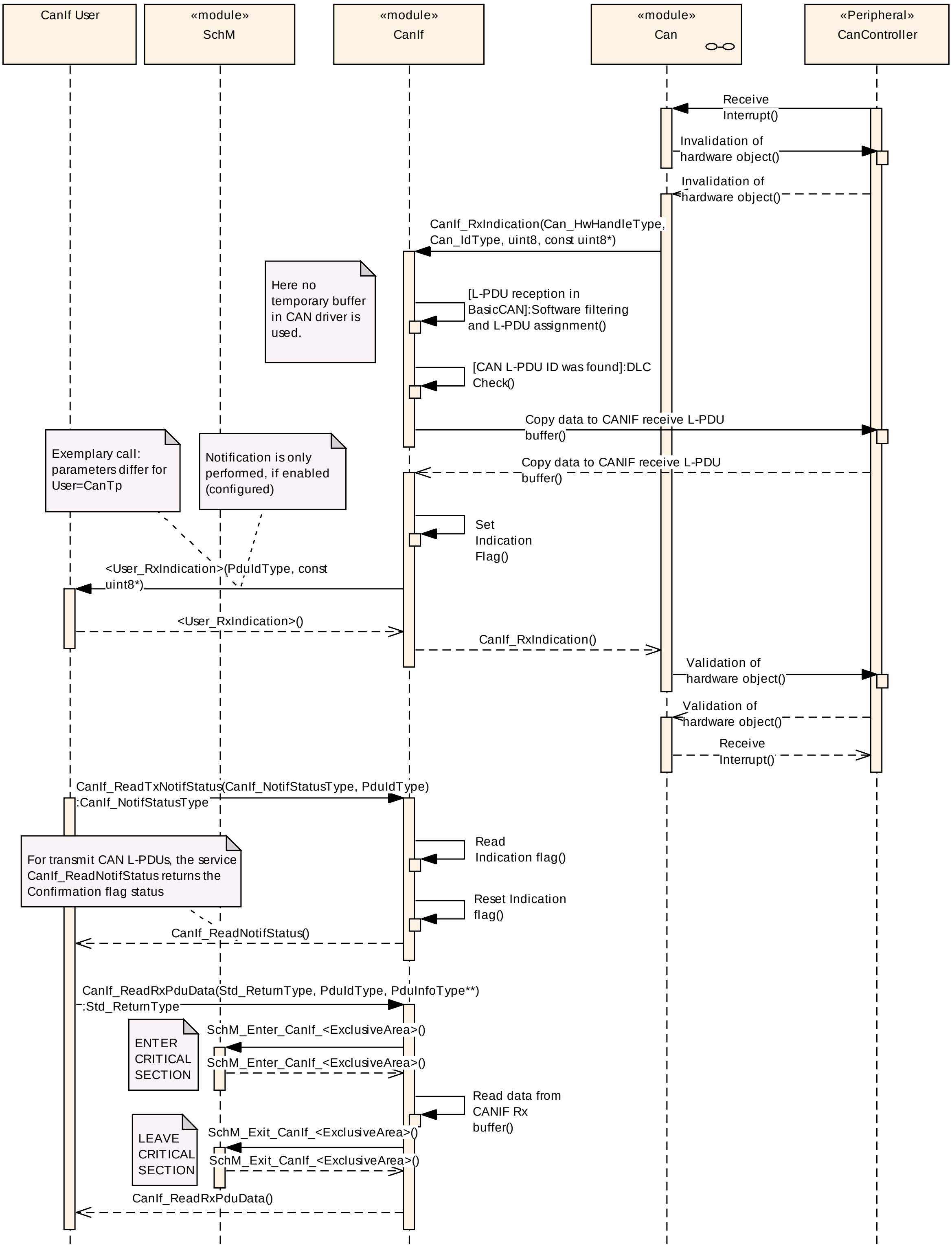

CanIf User
«module» SchM
«module» CanIf
«module» Can
«Peripheral» CanController
Receive Interrupt()
Invalidation of hardware object()
Invalidation of hardware object()
CanIf_RxIndication(Can_HwHandleType, Can_IdType, uint8, const uint8*)
Here no temporary buffer in CAN driver is used.
[L-PDU reception in BasicCAN]:Software filtering and L-PDU assignment()
[CAN L-PDU ID was found]:DLC Check()
Copy data to CANIF receive L-PDU buffer()
Exemplary call: parameters differ for User=CanTp
Notification is only performed, if enabled (configured)
Copy data to CANIF receive L-PDU buffer()
Set Indication Flag()
<User_RxIndication>(PduIdType, const uint8*)
<User_RxIndication>()
CanIf_RxIndication()
Validation of hardware object()
Validation of hardware object()
Receive Interrupt()
CanIf_ReadTxNotifStatus(CanIf_NotifStatusType, PduIdType) :CanIf_NotifStatusType
Read Indication flag()
For transmit CAN L-PDUs, the service CanIf_ReadNotifStatus returns the Confirmation flag status
Reset Indication flag()
CanIf_ReadNotifStatus()
CanIf_ReadRxPduData(Std_ReturnType, PduIdType, PduInfoType**) :Std_ReturnType
SchM_Enter_CanIf_<ExclusiveArea>()
ENTER CRITICAL SECTION
SchM_Enter_CanIf_<ExclusiveArea>()
Read data from CANIF Rx buffer()
SchM_Exit_CanIf_<ExclusiveArea>()
LEAVE CRITICAL SECTION
SchM_Exit_CanIf_<ExclusiveArea>()
CanIf_ReadRxPduData()

Figure 28 Read received data

Activity	*Description*
Receive Interrupt	The CAN controller signals a successful reception and triggers a receive interrupt.
Invalidation of CAN hardware object,	The CPU (CAN Driver) get exclusive access rights to the

provide CPU access to CAN mailbox	CAN mailbox or at least to the corresponding hardware object, where new data were received.
Buffering, normalizing	The L-SDU is normalized and is buffered in the temporary buffer located in the CAN Driver. Each CAN Driver owns a temporary buffer for every physical channel only if normalizing of the data is necessary.
Indication to **CAN Interface**	The reception is indicated to the CAN Interface by calling of `CanIf_RxIndication()`. The HRH specifies the CAN RAM hardware object and the corresponding CAN controller, which contains the received L-PDU. The temporary buffer is referenced to the CAN Interface by *CanSduPtr.
Software Filtering	The Software Filtering checks, whether the received L-PDU will be processed on a local ECU. If not, the received L-PDU is not indicated to upper layers. Further processing is suppressed.
DLC check	If the L-PDU is found, the DLC of the received L-PDU is compared with the expected, statically configured one for the received L-PDU.
Copy data	The data is copied out of the CAN hardware into the receive CAN L-PDU buffers in the CAN Interface. During access the CAN hardware buffers must be unlocked for CPU access/locked fro CAN controller access.
Indication Flag	Set indication status flag for the received L-PDU in the CAN Interface.
Receive Indication to the upper layer	The corresponding receive indication service of the upper layer is called. This signals a successful reception to the target upper layer. The parameter CanPduId specifies the L-PDU, the second parameter is the reference on the temporary buffer within the L-SDU.
Validation of CAN hardware object, allow access of CAN controller to CAN mailbox	The CAN controller get back exclusive access rights to the CAN mailbox or at least to the corresponding hardware object, where new data were already being copied into the upper layer buffer.
Read indication status	Times later the upper layer can read the indication status by call of `CanIf_ReadRxNotifStatus()`. This service can also be used for transmit L-PDUs. Then it return the confirmation status.
Reset indication status	Before `CanIf_ReadRxNotifStatus()` returns, the indication status is reset.
Read received data	Times later the upper layer can read the received data by call of `CanIf_ReadRxNotifStatus()`.

9.11 Start CAN network

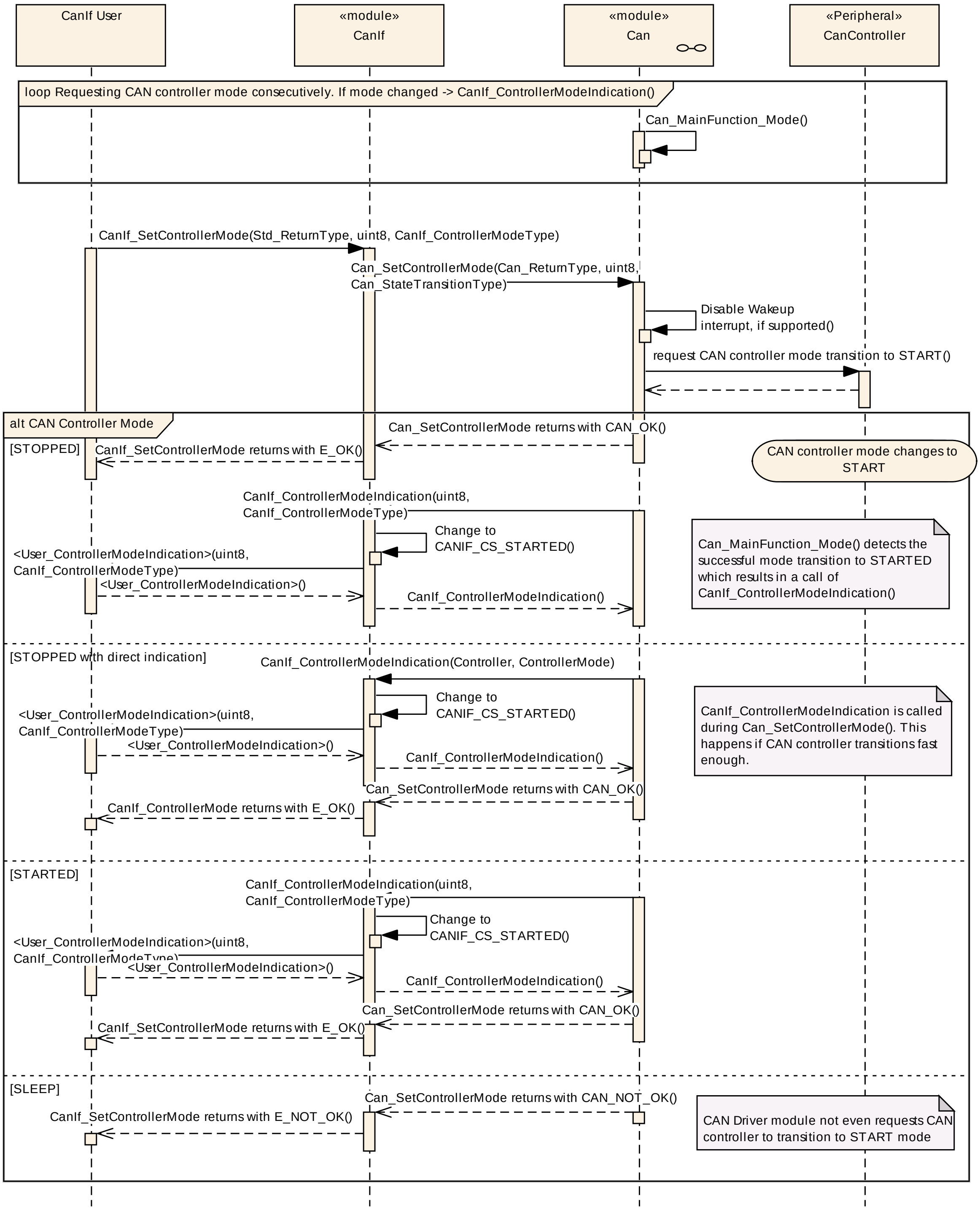

CanIf User
«module» CanIf
«module» Can
«Peripheral» CanController
loop Requesting CAN controller mode consecutively. If mode changed -> CanIf_ControllerModeIndication()
Can_MainFunction_Mode()
CanIf_SetControllerMode(Std_ReturnType, uint8, CanIf_ControllerModeType)
Can_SetControllerMode(Can_ReturnType, uint8, Can_StateTransitionType)
Disable Wakeup interrupt, if supported()
request CAN controller mode transition to START()
alt CAN Controller Mode
[STOPPED]
Can_SetControllerMode returns with CAN_OK()
CanIf_SetControllerMode returns with E_OK()
CAN controller mode changes to START
CanIf_ControllerModeIndication(uint8, CanIf_ControllerModeType)
Change to CANIF_CS_STARTED()
<User_ControllerModeIndication>(uint8, CanIf_ControllerModeType)
<User_ControllerModeIndication>()
CanIf_ControllerModeIndication()
Can_MainFunction_Mode() detects the successful mode transition to STARTED which results in a call of CanIf_ControllerModeIndication()
[STOPPED with direct indication]
CanIf_ControllerModeIndication(Controller, ControllerMode)
Change to CANIF_CS_STARTED()
<User_ControllerModeIndication>(uint8, CanIf_ControllerModeType)
<User_ControllerModeIndication>()
CanIf_ControllerModeIndication()
CanIf_ControllerModeIndication is called during Can_SetControllerMode(). This happens if CAN controller transitions fast enough.
Can_SetControllerMode returns with CAN_OK()
CanIf_ControllerMode returns with E_OK()
[STARTED]
CanIf_ControllerModeIndication(uint8, CanIf_ControllerModeType)
Change to CANIF_CS_STARTED()
<User_ControllerModeIndication>(uint8, CanIf_ControllerModeType)
<User_ControllerModeIndication>()
CanIf_ControllerModeIndication()
Can_SetControllerMode returns with CAN_OK()
CanIf_SetControllerMode returns with E_OK()
[SLEEP]
Can_SetControllerMode returns with CAN_NOT_OK()
CanIf_SetControllerMode returns with E_NOT_OK()
CAN Driver module not even requests CAN controller to transition to START mode

Figure 29 Start CAN network

This sequence diagram resembles “Stop CAN network” or “Sleep CAN network”.

Activity	***Description***

Loop requesting CAN controller mode consecutively.	The `Can_MainFunction_Mode()` is triggered consecutively. It checks the HW if a controller mode has changed. If so, it is notified via a function call of `CanIf_ControllerModeIndication(Controller, ControllerMode)`.
The upper layer requests "STARTED" mode of the desired CAN controller	The upper layer calls `CanIf_SetControllerMode (ControllerId, CANIF_CS_STARTED)` to request STARTED mode for the requested CAN controller.
CanDrv disables wake up interrupts, if supported	This is only done in case of requesting "STARTED" mode. If "SLEEP" mode of CAN controller is requested, here the wake up interrupts are enabled. In case of "STOPPED", nothing happens.
CanDrv requests the CAN controller to transition into the requested mode (CAN_T_START).	During function call `Can_SetControllerMode(Controller, Can_StateTransitionType)`, the CanDrv enters the request into the hardware of the CAN controller. This may mean that the controller mode transitions directly, but it could mean that it takes a few milliseconds until the controller changes its state. It depends on the controllers.
The following reaction depends on the controller and its current operation mode	
CAN controller was in STOPPED mode	The former request `Can_SetControllerMode()` returns and informs CanIf about a successful request which in turn returns the upper layer request `CanIf_SetControllerMode()`. The `Can_MainFunction_Mode()` detects the successful mode transition of the CAN controller and inform the CanIf asynchronously via `CanIf_ControllerModeIndication(Controller, CANIF_CS_STARTED)` . Then the CanIf updates its CCMSM mode.
CAN controller was in STOPPED mode and the CAN controller transitions very fast so that mode indication is called during transition request	During the former request `Can_SetControllerMode()` the function `CanIf_ControllerModeIndication(Controller, CANIF_CS_STARTED)` is called to inform the CanIf directly about the successful mode transition. Then the CanIf updates its CCMSM mode. When `CanIf_ControllerModeIndication(Controller, CANIF_CS_STARTED)` returned, the request `Can_SetControllerMode()` returns and informs CanIf about a successful request which in turn returns the upper layer request `CanIf_SetControllerMode()`.
CAN controller was in STARTED mode	During the former request `Can_SetControllerMode()` the function `CanIf_ControllerModeIndication(Controller, CANIF_CS_STARTED)` is called to inform the CanIf directly about the successful mode transition (because the mode was already started). Then the CanIf updates its CCMSM mode (not really necessary). When `CanIf_ControllerModeIndication(Controller, CANIF_CS_STARTED)` returned, the request `Can_SetControllerMode()` returns and informs CanIf about a successful request which in turn returns the upper layer request `CanIf_SetControllerMode()`.
CAN controller was in SLEEP mode	This transition is not allowed -> CAN_NOT_OK and E_NOT_OK.

9.12 BusOff notification

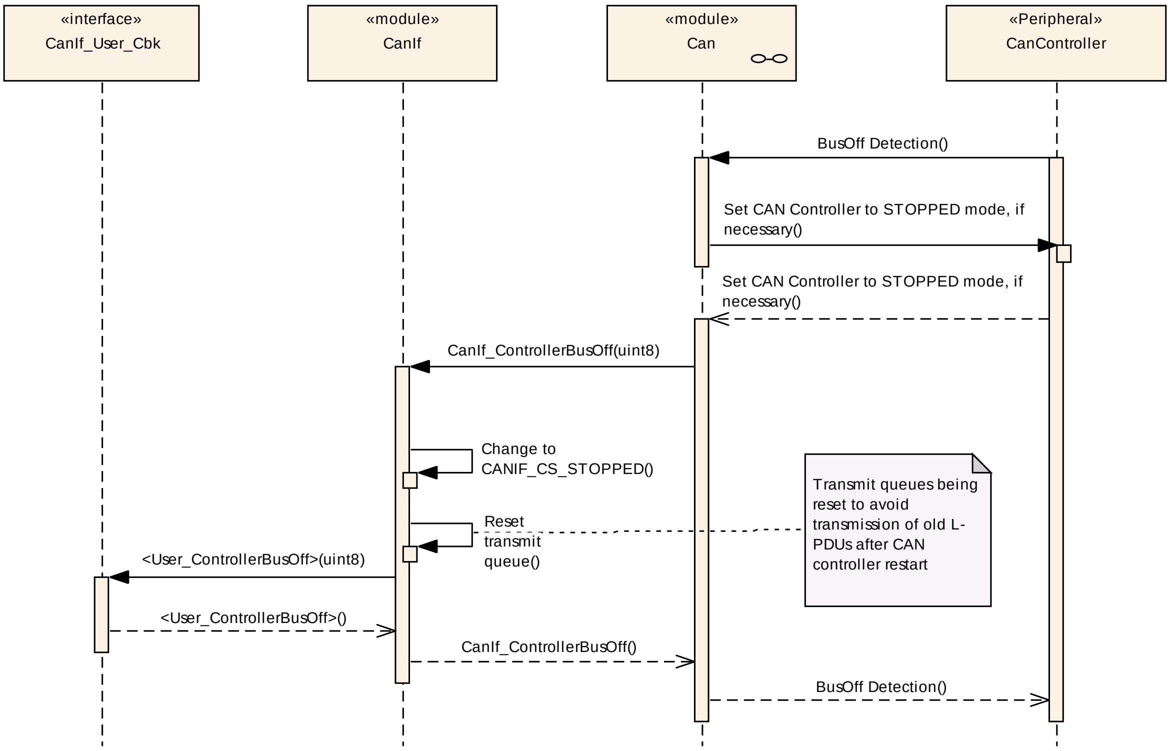

«interface» CanIf_User_Cbk
«module» CanIf
«module» Can
«Peripheral» CanController
BusOff Detection()
Set CAN Controller to STOPPED mode, if necessary()
Set CAN Controller to STOPPED mode, if necessary()
CanIf_ControllerBusOff(uint8)
Change to CANIF_CS_STOPPED()
Reset transmit queue()
Transmit queues being reset to avoid transmission of old L-PDUs after CAN controller restart
<User_ControllerBusOff>(uint8)
<User_ControllerBusOff>()
CanIf_ControllerBusOff()
BusOff Detection()

Figure 30 BusOff notification

Activity	*Description*
BusOff detection interrupt	The CAN controller signals a BusOff event.
Stop CAN controller	CAN controller is set to STOPPED mode by the CAN Driver, if necessary.
BusOff indication to CAN Interface	BusOff is notified to the CanIf by calling of `CanIf_ControllerBusOff()`
BusOff indication to upper layer (CanSM)	BusOff is notified to the upper layer by calling of `<User_ControllerBusOff>()`

9.13 BusOff recovery

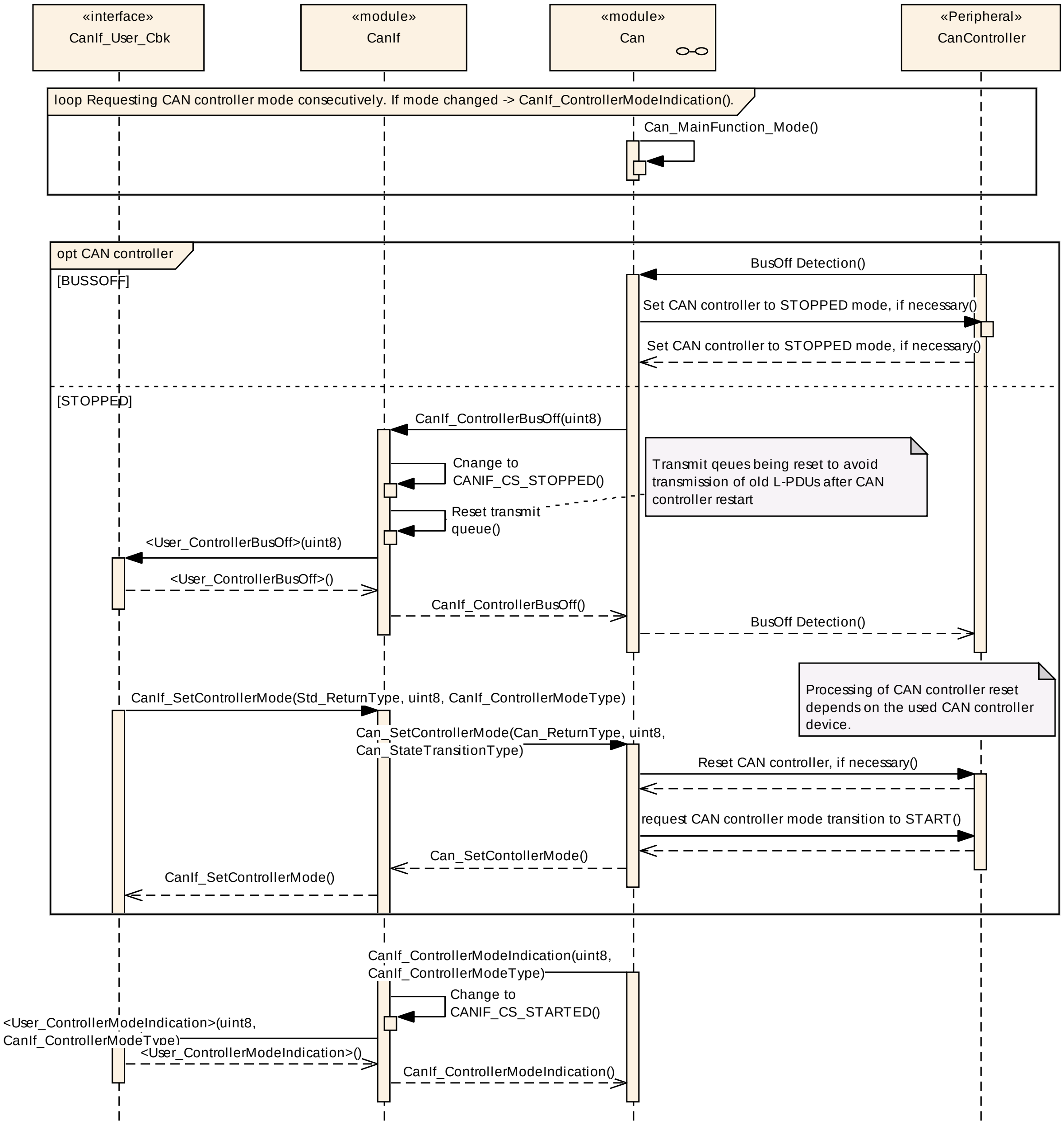

«interface»
CanIf_User_Cbk
«module»
CanIf
«module»
Can
«Peripheral»
CanController
loop Requesting CAN controller mode consecutively. If mode changed -> CanIf_ControllerModeIndication().
Can_MainFunction_Mode()
opt CAN controller
[BUSSOFF]
BusOff Detection()
Set CAN controller to STOPPED mode, if necessary()
Set CAN controller to STOPPED mode, if necessary()
[STOPPED]
CanIf_ControllerBusOff(uint8)
Cnange to CANIF_CS_STOPPED()
Transmit qeues being reset to avoid transmission of old L-PDUs after CAN controller restart
Reset transmit queue()
<User_ControllerBusOff>(uint8)
<User_ControllerBusOff>()
CanIf_ControllerBusOff()
BusOff Detection()
Processing of CAN controller reset depends on the used CAN controller device.
CanIf_SetControllerMode(Std_ReturnType, uint8, CanIf_ControllerModeType)
Can_SetControllerMode(Can_ReturnType, uint8, Can_StateTransitionType)
Reset CAN controller, if necessary()
request CAN controller mode transition to START()
Can_SetContollerMode()
CanIf_SetControllerMode()
CanIf_ControllerModeIndication(uint8, CanIf_ControllerModeType)
Change to CANIF_CS_STARTED()
<User_ControllerModeIndication>(uint8, CanIf_ControllerModeType)
<User_ControllerModeIndication>()
CanIf_ControllerModeIndication()

Figure 31 BusOff recovery

Activity	*Description*
BusOff detection interrupt	The CAN controller signals a BusOff event.
Stop CAN controller	CAN controller is set to STOPPED mode by the CanDrv, if necessary
BusOff indication to CAN Interface	BusOff is notified to the CanIf by calling of `CanIf_ControllerBusOff().`The transmit buffers inside the CanIf will be reset.
BusOff indication to upper layer	BusOff is notified to the upper layer by calling of `<User_ControllerBusOff>()`
Upper Layer (CanSM) initiates BusOff Recovery	After a time specified by the BusOff Recovery algorithm the Recovery process itself in initiated by

	`CanIf_SetControllerMode (ControllerId, CANIF_CS_STARTED).`
Restart of CAN controller	The driver restarts the CAN controller by call of `Can_SetControllerMode (Controller, CAN_T_STARTED).`

10 Configuration specification

In general, this chapter defines configuration parameters and their clustering into containers. In order to support the specification chapter 10.1 describes fundamentals. It also specifies a template (table) you shall use for the parameter specification. We intend to leave chapter 10.1 in the specification to guarantee comprehension.

Chapter 10.2 specifies the structure (containers) and the parameters of the CanIf.

10.1 How to read this chapter

In addition to this section, it is highly recommended to read the documents:

- [2] Layered Software Architecture
- [6] Specification of ECU Configuration
  This document describes the AUTOSAR configuration methodology and the AUTOSAR configuration meta model in detail.

The following is only a short survey of the topic and it will not replace the ECU Configuration Specification document.

10.1.1 Configuration and configuration parameters

Configuration parameters define the variability of the generic part(s) of an implementation of a module. This means that only generic or configurable module implementation can be adapted to the environment (software/hardware) in use during system and/or ECU configuration.

The configuration of parameters can be achieved at different times during the software process: before compile time, before link time or after build time. In the following, the term “configuration class” (of a parameter) shall be used in order to refer to a specific configuration point in time.

10.1.2 Variants

Variants describe sets of configuration parameters. E.g., variant 1: only pre-compile time configuration parameters; variant 2: mix of pre-compile- and post build time-configuration parameters. In one variant a parameter can only be of one configuration class.

10.1.3 Containers

Containers structure the set of configuration parameters. This means:

- *all* configuration parameters are kept in containers.
- (sub-) containers can reference (sub-) containers. It is possible to assign a multiplicity to these references. The multiplicity then defines the possible number of instances of the contained parameters.

10.1.4 Specification template for configuration parameters

The following tables consist of three sections:

- the general section

- the configuration parameter section
- the section of included/referenced containers

Pre-compile time - specifies whether the configuration parameter shall be of configuration class *Pre-compile time* or not

Label	***Description***
x	The configuration parameter shall be of configuration class *Pre-compile time*.
--	The configuration parameter shall never be of configuration class *Pre-compile time*.

Link time - specifies whether the configuration parameter shall be of configuration class *Link time* or not

Label	***Description***
x	The configuration parameter shall be of configuration class *Link time*.
--	The configuration parameter shall never be of configuration class *Link time*.

Post Build - specifies whether the configuration parameter shall be of configuration class *Post Build* or not

Label	***Description***
x	The configuration parameter shall be of configuration class *Post Build* and no specific implementation is required.
L	*Loadable* - the configuration parameter shall be of configuration class *Post Build* and only one configuration parameter set resides in the ECU.
M	*Multiple* - the configuration parameter shall be of configuration class *Post Build* and is selected out of a set of multiple parameters by passing a dedicated pointer to the init function of the module.
--	The configuration parameter shall never be of configuration class *Post Build*.

10.2 Containers and configuration parameters

The following chapters summarize all configuration parameters. The detailed meanings of the parameters describe chapter [7 Functional specification] and chapter [8 API specification].

[CANIF104] ⌈The listed configuration items can be derived from a network description database, which is based on the EcuConfigurationTemplate. The configuration tool shall extract all information to configure the CanIf. ⌋(BSW01015)

[CANIF131] ⌈The consistency of the configuration must be checked by the configuration tool at configuration time. Configuration rules and constraints for plausibility checks shall be performed during configuration time, where possible.⌋()

[CANIF066] ⌈The CanIf has access to the CanDrv configuration data. All public CanDrv configuration data are described in [8] Specification of CAN Driver.⌋()

[CANIF132] ⌈These dependencies between CanDrv and CanIf configuration must be provided at configuration time by the configuration tools.⌋()

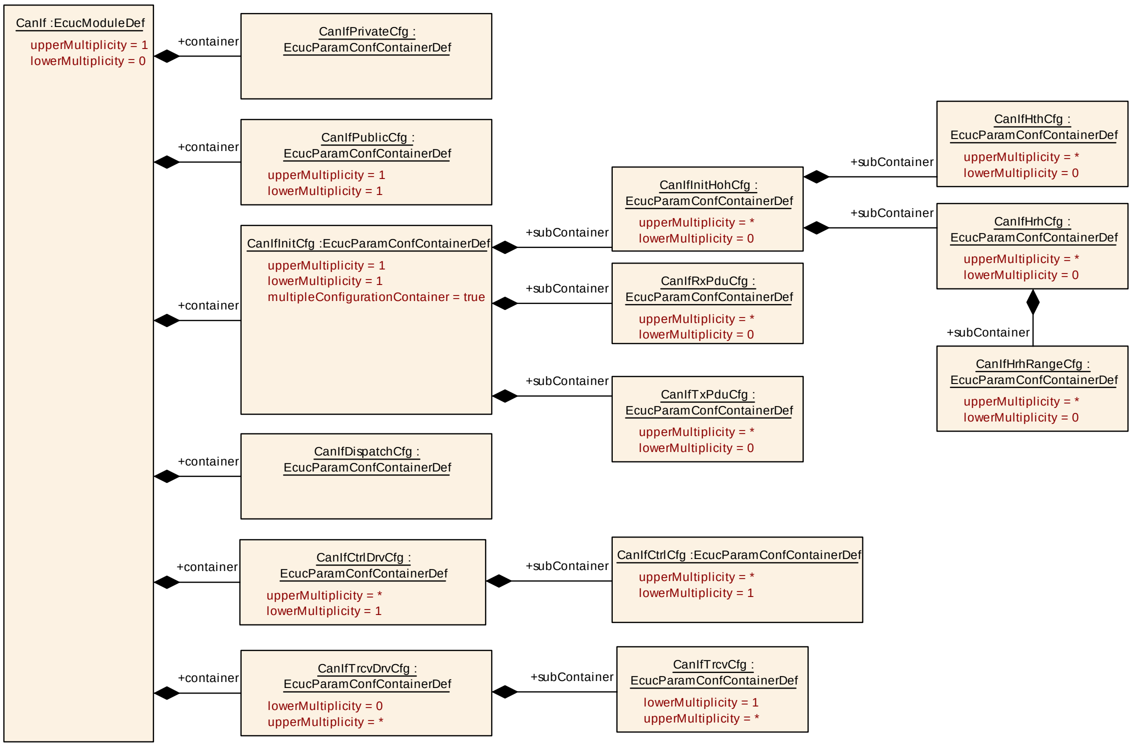


Figure 32 Overview about CAN Interface configuration containers

10.2.1 Variants

[CANIF460] ⌈Variant 1: Only pre compile time parameters. ⌋()

[CANIF461] ⌈Variant 2: Mix of pre compile- and link time parameters. ⌋ (BSW00344)

[CANIF462] ⌈Variant 3: Mix of pre compile-, link time and post build time parameters.⌋(BSW00344, BSW00404, BSW00342)

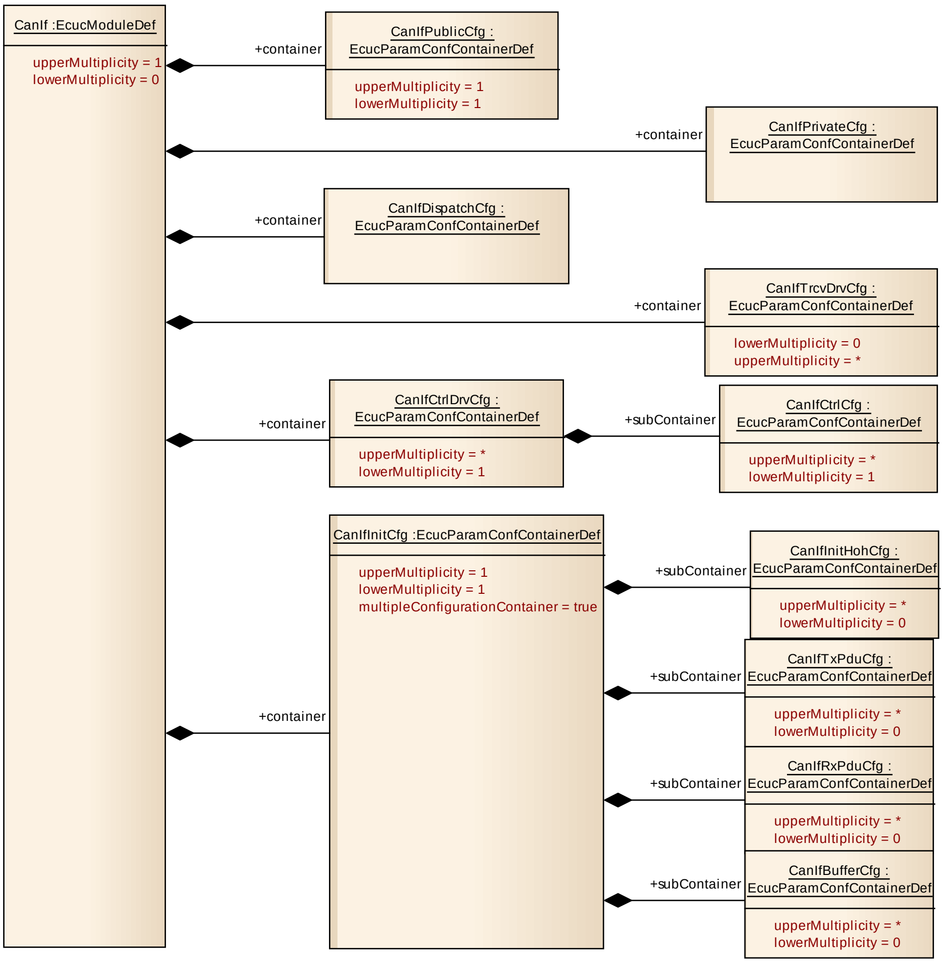
10.2.2 CanIf

SWS Item	**CANIF244_Conf :**
Module Name	*CanIf*
Module Description	This container includes all necessary configuration sub-containers according the CAN Interface configuration structure.

Included Containers

Container Name	*Multiplicity*	*Scope / Dependency*
CanIfCtrlDrvCfg	1..*	Configuration parameters for all the underlying CAN Driver modules are aggregated under this container. For each CAN Driver module a seperate instance of this container has to be provided.
CanIfDispatchCfg	1	Callback functions provided by upper layer modules of the CanIf. The callback functions defined in this container are common to all configured CAN Driver / CAN Transceiver Driver modules.
CanIfInitCfg	1	This container contains the init parameters of the CAN Interface. At least one (if only on CanIf with one possible Configuration), but multiple (CanIf with different Configurations) instances of this container are possible.
CanIfPrivateCfg	1	This container contains the private configuration (parameters) of the CAN Interface.
CanIfPublicCfg	1	This container contains the public configuration (parameters) of the CAN Interface.
CanIfTrcvDrvCfg	0..*	This container contains the configuration (parameters) of all addressed CAN transceivers by each underlying CAN Transceiver Driver module. For each CAN transceiver Driver a seperate instance of this container shall be provided.

CANIF244_Conf (This SWS Item ID belongs to the table above. The generated Artifact is faulty.)



10.2.3 CanIfPrivateCfg

SWS Item	**CANIF245_Conf :**
Container Name	CanIfPrivateCfg{CanInterfacePrivateConfiguration}
Description	This container contains the private configuration (parameters) of the CAN Interface.
Configuration Parameters	

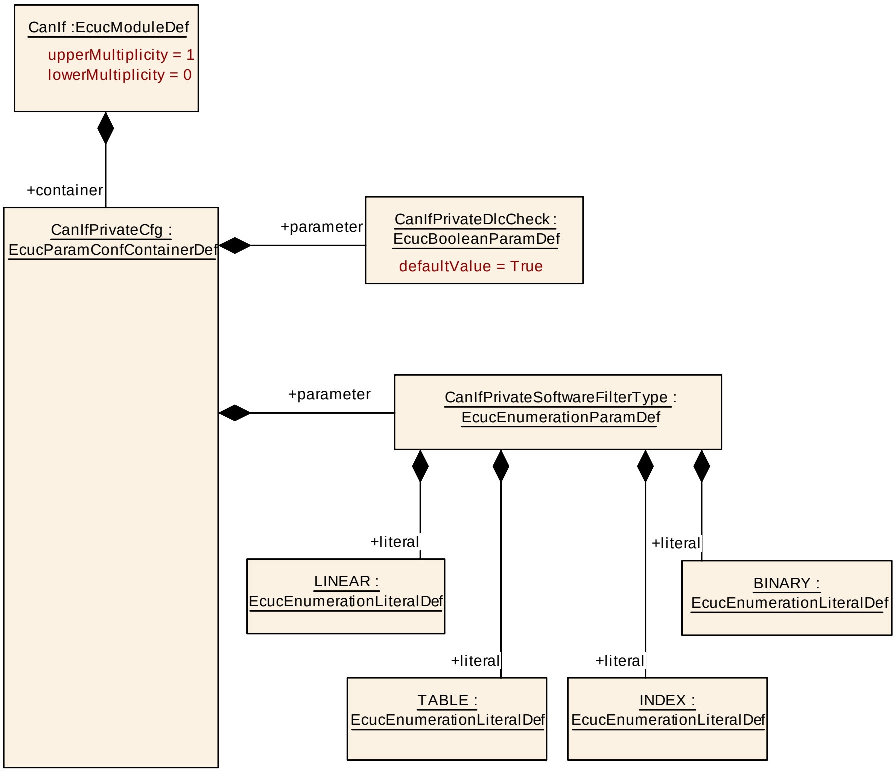
SWS Item	**CANIF617_Conf :**		
Name	CanIfPrivateDlcCheck {CANIF_PRIVATE_DLC_CHECK}		
Description	Selects whether the DLC check is supported. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	true		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Module		

SWS Item	**CANIF619_Conf :**		
Name	CanIfPrivateSoftwareFilterType {CANIF_PRIVATE_SOFTWARE_FILTER_TYPE}		
Description	Selects the desired software filter mechanism for reception only. Each implemented software filtering method is identified by this enumeration number. Range: Types implemented software filtering methods		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	BINARY	Selects Binary Filter method.	
	INDEX	Selects Index Filter method.	
	LINEAR	Selects Linear Filter method.	
	TABLE	Selects Table Filter method.	
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Module dependency: BasicCAN reception must be enabled by referenced parameter CAN_HANDLE_TYPE of the CAN Driver module via CANIF_HRH_HANDLETYPE_REF for at least one HRH.		

SWS Item	**CANIF675_Conf :**		
Name	CanIfSupportTTCAN		
Description	Defines whether TTCAN is supported. TRUE: TTCAN is supported. FALSE: TTCAN is not supported, only normal CAN communication is possible.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanIfTTGenera	0..1	This container is only included and valid if TTCAN Interface SWS is used

l		and TTCAN is enabled. This container contains the parameters, which define if and in which way TTCAN is supported. CanIfTTGeneral is only included, if the controller supports TTCAN.


CanIf :EcucModuleDef
upperMultiplicity = 1
lowerMultiplicity = 0
+container
CanIfPrivateCfg : EcucParamConfContainerDef
+parameter
CanIfPrivateDlcCheck : EcucBooleanParamDef
defaultValue = True
+parameter
CanIfPrivateSoftwareFilterType : EcucEnumerationParamDef
+literal
LINEAR : EcucEnumerationLiteralDef
+literal
BINARY : EcucEnumerationLiteralDef
+literal
TABLE : EcucEnumerationLiteralDef
+literal
INDEX : EcucEnumerationLiteralDef

10.2.4 CanIfPublicCfg

SWS Item	**CANIF246_Conf :**
Container Name	CanIfPublicCfg{CanInterfacePublicConfiguration}
Description	This container contains the public configuration (parameters) of the CAN Interface.
Configuration Parameters	

SWS Item	**CANIF522_Conf :**		
Name	CanIfPublicCancelTransmitSupport {CANIF_PUBLIC_CANCEL_TRANSMIT_SUPPORT}		
Description	Configuration parameter to enable/disable dummy API for upper layer modules which allows to request the cancellation of an I-PDU.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**CANIF671_Conf :**
Name	CanIfPublicCddHeaderFile {CANIF_PUBLIC_CDD_HEADERFILE}
Description	Defines header files for callback functions which shall be included in

	case of CDDs. Range of characters is 1.. 32.		
Multiplicity	0..*		
Type	EcucStringParamDef		
Default value	--		
maxLength	32		
minLength	1		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**CANIF785_Conf :**		
Name	CanIfPublicChangeBaudrateSupport {CANIF_PUBLIC_CHANGE_BAUDRATE_SUPPORT}		
Description	Configuration parameter to enable/disable the API to change the baudrate of a CAN controller. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**CANIF614_Conf :**		
Name	CanIfPublicDevErrorDetect {CANIF_PUBLIC_DEV_ERROR_DETECT}		
Description	Enables and disables the development error detection and notification mechanism. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	true		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Module		

SWS Item	**CANIF742_Conf :**		
Name	CanIfPublicHandleTypeEnum {CANIF_PUBLIC_HANDLE_TYPE_ENUM}		
Description	This parameter is used to configure the Can_HwHandleType. The Can_HwHandleType represents the hardware object handles of a CAN hardware unit. For CAN hardware units with more than 255 HW objects the extended range shall be used (UINT16).		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	UINT16	--	
	UINT8	--	
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: CAN stack dependency: Can_HwHandleType		

SWS Item	**CANIF612_Conf :**

Name	CanIfPublicMultipleDrvSupport {CANIF_PUBLIC_MULTIPLE_DRV_SUPPORT}		
Description	Selects support for multiple CAN Drivers. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	true		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**CANIF615_Conf :**		
Name	CanIfPublicNumberOfCanHwUnits {CANIF_PUBLIC_NUMBER_OF_CAN_HW_UNITS}		
Description	Number of served CAN hardware units.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	1 .. 255		
Default value	1		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**CANIF772_Conf :**		
Name	CanIfPublicPnSupport {CANIF_PUBLIC_PN_SUPPORT}		
Description	Selects support of Partial Network features in CanIf. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: COM Stack		

SWS Item	**CANIF607_Conf :**		
Name	CanIfPublicReadRxPduDataApi {CANIF_PUBLIC_READRXPDU_DATA_API}		
Description	Enables / Disables the API CanIf_ReadRxPduData() for reading received L-PDU data. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**CANIF608_Conf :**
Name	CanIfPublicReadRxPduNotifyStatusApi {CANIF_PUBLIC_READRXPDU_NOTIFY_STATUS_API}
Description	Enables and disables the API for reading the received L-PDU data. True: Enabled False: Disabled
Multiplicity	1

Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**CANIF609_Conf :**		
Name	CanIfPublicReadTxPduNotifyStatusApi {CANIF_PUBLIC_READTXPDU_NOTIFY_STATUS_API}		
Description	Enables and disables the API for reading the notification status of transmit and receive L-PDUs. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**CANIF610_Conf :**		
Name	CanIfPublicSetDynamicTxIdApi {CANIF_PUBLIC_SETDYNAMICTXID_API}		
Description	Enables and disables the API for reconfiguration of the CAN Identifier for each Transmit L-PDU. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**CANIF618_Conf :**		
Name	CanIfPublicTxBuffering {CANIF_PUBLIC_TX_BUFFERING}		
Description	Enables and disables the buffering of transmit L-PDUs (rejected by the CanDrv) within the CAN Interface module. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: CAN stack		

SWS Item	**CANIF733_Conf :**		
Name	CanIfPublicTxConfirmPollingSupport {CANIF_PUBLIC_TXCONFIRM_POLLING_SUPPORT}		
Description	Configuration parameter to enable/disable the API to poll for Tx Confirmation state.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	

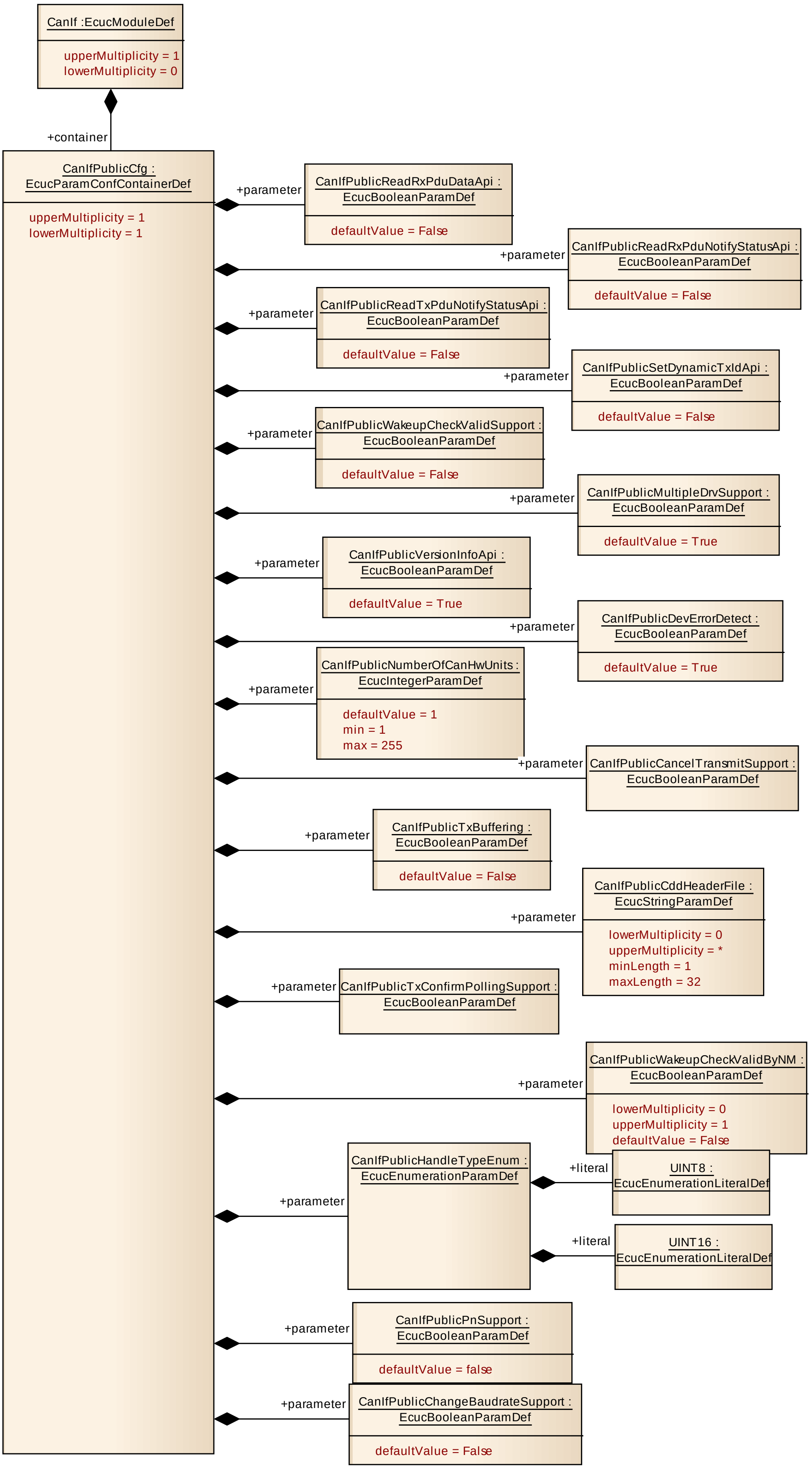
	Post-build time	--	
Scope / Dependency	scope: CanIf module dependency: CAN State Manager module		

SWS Item	**CANIF613_Conf :**		
Name	CanIfPublicVersionInfoApi {CANIF_PUBLIC_VERSION_INFO_API}		
Description	Enables and disables the API for reading the version information about the CAN Interface. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	true		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**CANIF741_Conf :**		
Name	CanIfPublicWakeupCheckValidByNM {CANIF_PUBLIC_WAKEUP_CHECK_VALID_BY_NM}		
Description	If enabled, only NM messages shall validate a detected wake-up event (see CANIF722) at the corresponding wake-up source in the CanIf. If disabled, all messages shall validate such a wake-up event. This parameter depends on CANIF_PUBLIC_WAKEUP_CHECK_VALID_API and shall only be configurable, if it is enabled. True: Enabled False: Disabled		
Multiplicity	0..1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: CANIF_PUBLIC_WAKEUP_CHECK_VALID_API		

SWS Item	**CANIF611_Conf :**		
Name	CanIfPublicWakeupCheckValidSupport {CANIF_PUBLIC_WAKEUP_CHECK_VALIDATION_SUPPORT}		
Description	Selects support for wake up validation True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

No Included Containers

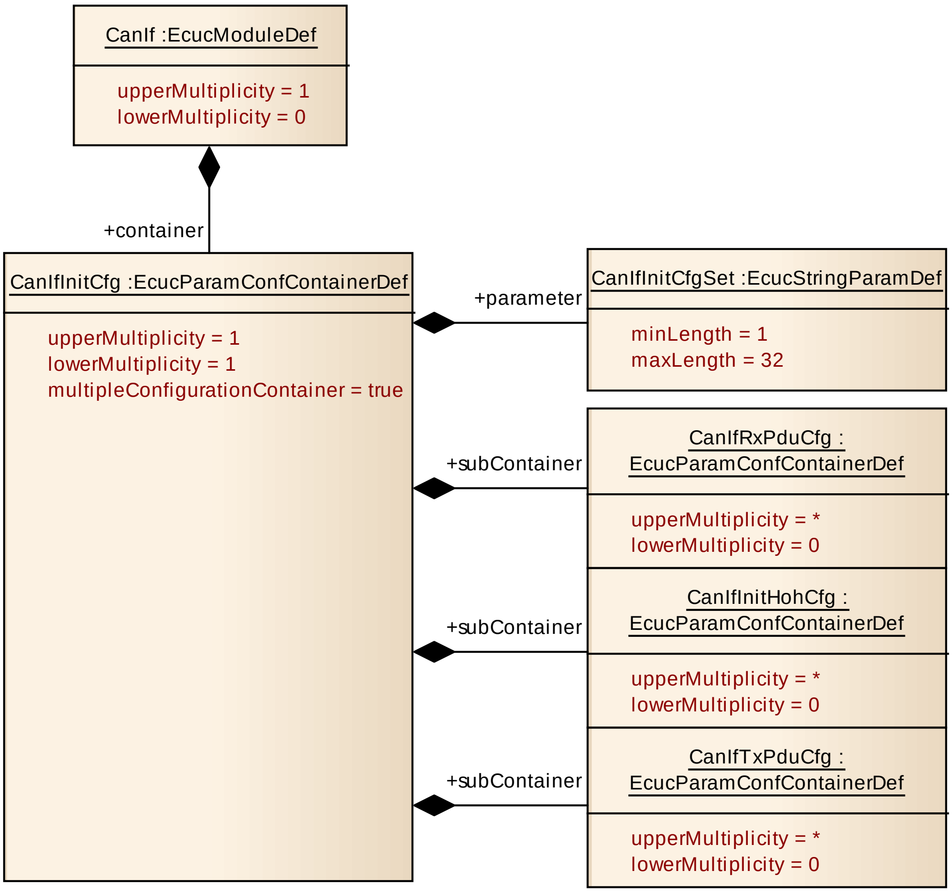

CanIf :EcucModuleDef
upperMultiplicity = 1
lowerMultiplicity = 0
+container
CanIfPublicCfg :
EcucParamConfContainerDef
upperMultiplicity = 1
lowerMultiplicity = 1
+parameter
CanIfPublicReadRxPduDataApi :
EcucBooleanParamDef
defaultValue = False
+parameter
CanIfPublicReadRxPduNotifyStatusApi :
EcucBooleanParamDef
defaultValue = False
+parameter
CanIfPublicReadTxPduNotifyStatusApi :
EcucBooleanParamDef
defaultValue = False
+parameter
CanIfPublicSetDynamicTxIdApi :
EcucBooleanParamDef
defaultValue = False
+parameter
CanIfPublicWakeupCheckValidSupport :
EcucBooleanParamDef
defaultValue = False
+parameter
CanIfPublicMultipleDrvSupport :
EcucBooleanParamDef
defaultValue = True
+parameter
CanIfPublicVersionInfoApi :
EcucBooleanParamDef
defaultValue = True
+parameter
CanIfPublicDevErrorDetect :
EcucBooleanParamDef
defaultValue = True
+parameter
CanIfPublicNumberOfCanHwUnits :
EcucIntegerParamDef
defaultValue = 1
min = 1
max = 255
+parameter
CanIfPublicCancelTransmitSupport :
EcucBooleanParamDef
+parameter
CanIfPublicTxBuffering :
EcucBooleanParamDef
defaultValue = False
+parameter
CanIfPublicCddHeaderFile :
EcucStringParamDef
lowerMultiplicity = 0
upperMultiplicity = *
minLength = 1
maxLength = 32
+parameter
CanIfPublicTxConfirmPollingSupport :
EcucBooleanParamDef
+parameter
CanIfPublicWakeupCheckValidByNM :
EcucBooleanParamDef
lowerMultiplicity = 0
upperMultiplicity = 1
defaultValue = False
+parameter
CanIfPublicHandleTypeEnum :
EcucEnumerationParamDef
+literal
UINT8 :
EcucEnumerationLiteralDef
+literal
UINT16 :
EcucEnumerationLiteralDef
+parameter
CanIfPublicPnSupport :
EcucBooleanParamDef
defaultValue = false
+parameter
CanIfPublicChangeBaudrateSupport :
EcucBooleanParamDef
defaultValue = False

10.2.5 CanIfInitCfg

SWS Item	**CANIF247_Conf :**
Container Name	CanIfInitCfg{CanInterfaceInitConfiguration} [Multi Config Container]
Description	This container contains the init parameters of the CAN Interface. At least one (if only on CanIf with one possible Configuration), but multiple (CanIf with different Configurations) instances of this container are possible.
Configuration Parameters	

SWS Item	**CANIF623_Conf :**		
Name	CanIfInitCfgSet {CANIF_INIT_CONFIGSET}		
Description	Selects the CAN Interface specific configuration setup. This type of the external data structure shall contain the post build initialization data for the CAN Interface for all underlying CAN Dirvers. constant to CanIf_ConfigType		
Multiplicity	1		
Type	EcucStringParamDef		
Default value	--		
maxLength	32		
minLength	1		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Module		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanIfBufferCfg	0..*	This container contains the Txbuffer configuration. Multiple buffers with different sizes could be configured. If CanIfBufferSize (CANIF834_Conf) equals 0, the CanIf Tx L-PDU only refers via this CanIfBufferCfg the corresponding CanIfHthCfg.
CanIfInitHohCfg	0..*	This container contains the references to the configuration setup of each underlying CAN Driver.
CanIfRxPduCfg	0..*	This container contains the configuration (parameters) of each receive CAN L-PDU. The SHORT-NAME of "CanIfRxPduConfig" container itself represents the symolic name of Receive L-PDU.
CanIfTxPduCfg	0..*	This container contains the configuration (parameters) of a transmit CAN L-PDU. It has to be configured as often as a transmit CAN L-PDU is needed. The SHORT-NAME of "CanIfTxPduConfig" container represents the symolic name of Transmit L-PDU.



10.2.6 CanIfTxPduCfg

SWS Item	**CANIF248_Conf :**
Container Name	CanIfTxPduCfg{CANIF_INIT_TX_PDU_CFG}
Description	This container contains the configuration (parameters) of a transmit CAN L-PDU. It has to be configured as often as a transmit CAN L-PDU is needed. The SHORT-NAME of "CanIfTxPduConfig" container represents the symolic name of Transmit L-PDU.
Configuration Parameters	

SWS Item	**CANIF592_Conf :**		
Name	CanIfTxPduCanId {CANIF_TXPDU_CANID}		
Description	CAN Identifier of transmit CAN L-PDUs used by the CAN Driver for CAN L-PDU transmission. Range: 11 Bit For Standard CAN Identifier ... 29 Bit For Extended CAN identifier		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 536870911		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME

	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Network		

SWS Item	**CANIF590_Conf :**		
Name	CanIfTxPduCanIdType {CANIF_TXPDU_CANIDTYPE}		
Description	Type of CAN Identifier of the transmit CAN L-PDU used by the CAN Driver module for CAN L-PDU transmission.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	EXTENDED_CAN	The CANID is of type Extended (29 bits)	
	STANDARD_CAN	The CANID is of type Standard (11 bits)	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Network		

SWS Item	**CANIF594_Conf :**		
Name	CanIfTxPduDlc {CANIF_TXPDU_DLC}		
Description	Data length code (in bytes) of transmit CAN L-PDUs used by the CAN Driver for CAN L-PDU transmission. The data area size of a CAN L-Pdu can have a range from 0 to 8 bytes.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 8		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Network		

SWS Item	**CANIF591_Conf :**		
Name	CanIfTxPduId {CANIF_TXPDU_ID}		
Description	ECU wide unique, symbolic handle for transmit CAN L-PDU. The CanIfTxPduId is configurable at pre-compile and post-built time. Range: 0..max. number of CantTxPduIds		
Multiplicity	1		
Type	EcucIntegerParamDef (Symbolic Name generated for this parameter)		
Range	0 .. 4294967295		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

SWS Item	**CANIF773_Conf :**
Name	CanIfTxPduPnFilterPdu {CANIF_TXPDU_PNFILTERPDU}
Description	If CanIfPublicPnFilterSupport is enabled, by this parameter PDUs could be configured which will pass the CanIfPnFilter. If

	there is no CanIfTxPduPnFilterPdu configured per controller, the corresponding controller applies no CanIfPnFilter.		
Multiplicity	0..1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	dependency: This parameter shall only be configurable if CanIfPublicPnSupport equals True.		

SWS Item	**CANIF589_Conf :**		
Name	CanIfTxPduReadNotifyStatus {CANIF_TXPDU_READ_NOTIFYSTATUS}		
Description	Enables and disables transmit confirmation for each transmit CAN L-PDU for reading its notification status. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Module dependency: CANIF_READTXPDU_NOTIFY_STATUS_API must be enabled.		

SWS Item	**CANIF593_Conf :**		
Name	CanIfTxPduType {CANIF_TXPDU_TYPE}		
Description	Defines the type of each transmit CAN L-PDU.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	DYNAMIC	CAN ID is defined at runtime.	
	STATIC	CAN ID is defined at compile-time.	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

SWS Item	**CANIF528_Conf :**		
Name	CanIfTxPduUserTxConfirmationName {CANIF_TXPDU_USERTXCONFIRMATION_NAME}		
Description	This parameter defines the name of the <User_TxConfirmation>. This parameter depends on the parameter CANIF_TXPDU_USERTXCONFIRMATION_UL. If CANIF_TXPDU_USERTXCONFIRMATION_UL equals CAN_TP, CAN_NM, PDUR, XCP or J1939TP, the name of the <User_TxConfirmation> is fixed. If CANIF_TXPDU_USERTXCONFIRMATION_UL equals CDD, the name of the <User_TxConfirmation> is selectable.		
Multiplicity	0..1		
Type	EcucFunctionNameDef		
Default value	--		
maxLength	32		
minLength	1		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE

	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**CANIF527_Conf :**		
Name	CanIfTxPduUserTxConfirmationUL {CANIF_TXPDU_USERTXCONFIRMATION_UL}		
Description	This parameter defines the upper layer (UL) module to which the confirmation of the successfully transmitted CANTXPDUID has to be routed via the <User_TxConfirmation>. This <User_TxConfirmation> has to be invoked when the confirmation of the configured CANTXPDUID will be received by a Tx confirmation event from the CAN Driver module. If no upper layer (UL) module is configured, no <User_TxConfirmation> has to be called in case of a Tx confirmation event of the CANTXPDUID from the CAN Driver module.		
Multiplicity	0..1		
Type	EcucEnumerationParamDef		
Range	CAN_NM	CAN NM	
	CAN_TP	CAN TP	
	CDD	Complex Device Driver	
	J1939TP	J1939Tp	
	PDUR	PDU Router	
	XCP	Extended Calibration Protocol	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

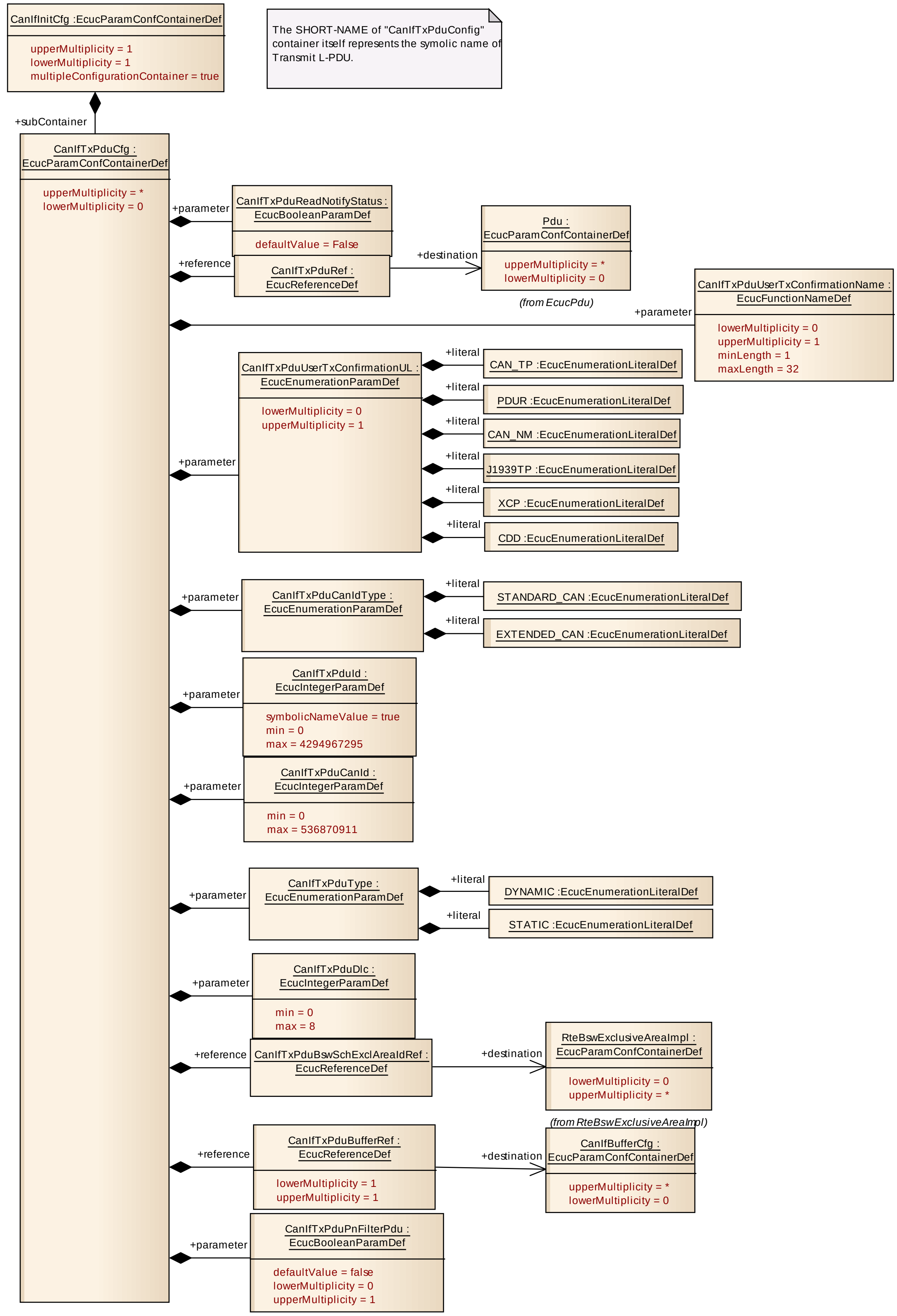
SWS Item	**CANIF670_Conf :**		
Name	CanIfTxPduBswSchExclAreaIdRef {CANIF_RXPDU_BSWSCH_EXCLAREAID_REF}		
Description	Reference to an exclusive area Id defined within the BSW Scheduler.		
Multiplicity	1		
Type	Reference to [RteBswExclusiveAreaImpl]		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**CANIF831_Conf :**		
Name	CanIfTxPduBufferRef {CANIF_TX_PDU_BUFFER_REF}		
Description	Configurable reference to a CanIf buffer configuration.		
Multiplicity	1		
Type	Reference to [CanIfBufferCfg]		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency			

SWS Item	**CANIF603_Conf :**
Name	CanIfTxPduRef {CANIF_TXPDU_REF}
Description	Reference to the "global" Pdu structure to allow harmonization of handle IDs in the COM-Stack.
Multiplicity	1
Type	Reference to [Pdu]

ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	--	
Scope / Dependency			

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanIfTTTxFrameTriggering	0..1	This container is only included and valid if TTCAN Interface SWS is used and TTCAN is enabled. Frame trigger for TTCAN transmission. CanIfTTTxFrameTriggering is only included, if the controller supports TTCAN and a joblist is used.


CanIfInitCfg :EcucParamConfContainerDef
upperMultiplicity = 1
lowerMultiplicity = 1
multipleConfigurationContainer = true
The SHORT-NAME of "CanIfTxPduConfig" container itself represents the symolic name of Transmit L-PDU.
+subContainer
CanIfTxPduCfg : EcucParamConfContainerDef
upperMultiplicity = *
lowerMultiplicity = 0
+parameter
CanIfTxPduReadNotifyStatus : EcucBooleanParamDef
defaultValue = False
+reference
CanIfTxPduRef : EcucReferenceDef
+destination
Pdu : EcucParamConfContainerDef
upperMultiplicity = *
lowerMultiplicity = 0
(from EcucPdu)
+parameter
CanIfTxPduUserTxConfirmationName : EcucFunctionNameDef
lowerMultiplicity = 0
upperMultiplicity = 1
minLength = 1
maxLength = 32
+parameter
CanIfTxPduUserTxConfirmationUL : EcucEnumerationParamDef
lowerMultiplicity = 0
upperMultiplicity = 1
+literal
CAN_TP :EcucEnumerationLiteralDef
+literal
PDUR :EcucEnumerationLiteralDef
+literal
CAN_NM :EcucEnumerationLiteralDef
+literal
J1939TP :EcucEnumerationLiteralDef
+literal
XCP :EcucEnumerationLiteralDef
+literal
CDD :EcucEnumerationLiteralDef
+parameter
CanIfTxPduCanIdType : EcucEnumerationParamDef
+literal
STANDARD_CAN :EcucEnumerationLiteralDef
+literal
EXTENDED_CAN :EcucEnumerationLiteralDef
+parameter
CanIfTxPduId : EcucIntegerParamDef
symbolicNameValue = true
min = 0
max = 4294967295
+parameter
CanIfTxPduCanId : EcucIntegerParamDef
min = 0
max = 536870911
+parameter
CanIfTxPduType : EcucEnumerationParamDef
+literal
DYNAMIC :EcucEnumerationLiteralDef
+literal
STATIC :EcucEnumerationLiteralDef
+parameter
CanIfTxPduDlc : EcucIntegerParamDef
min = 0
max = 8
+reference
CanIfTxPduBswSchExclAreaIdRef : EcucReferenceDef
+destination
RteBswExclusiveAreaImpl : EcucParamConfContainerDef
lowerMultiplicity = 0
upperMultiplicity = *
(from RteBswExclusiveAreaImpl)
+reference
CanIfTxPduBufferRef : EcucReferenceDef
lowerMultiplicity = 1
upperMultiplicity = 1
+destination
CanIfBufferCfg : EcucParamConfContainerDef
upperMultiplicity = *
lowerMultiplicity = 0
+parameter
CanIfTxPduPnFilterPdu : EcucBooleanParamDef
defaultValue = false
lowerMultiplicity = 0
upperMultiplicity = 1

10.2.7 CanIfRxPduCfg

SWS Item	**CANIF249_Conf :**
Container Name	CanIfRxPduCfg{CANIF_INIT_RX_PDU_CFG}
Description	This container contains the configuration (parameters) of each receive CAN L-PDU. The SHORT-NAME of "CanIfRxPduConfig" container itself represents the symolic name of Receive L-PDU.
Configuration Parameters	

SWS Item	**CANIF598_Conf :**		
Name	CanIfRxPduCanId {CANIF_RXPDU_CANID}		
Description	CAN Identifier of Receive CAN L-PDUs used by the CAN Interface. Exa: Software Filtering. This parameter is used if exactly one Can Identifier is assigned to the Pdu. If a range is assigned then the CanIfRxPduCanIdRange parameter shall be used. Range: 11 Bit For Standard CAN Identifier ... 29 Bit For Extended CAN identifier		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 536870911		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Network		

SWS Item	**CANIF596_Conf :**		
Name	CanIfRxPduCanIdType {CANIF_RXPDUID_CANIDTYPE}		
Description	CAN Identifier of receive CAN L-PDUs used by the CAN Driver for CAN L-PDU reception.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	EXTENDED_CAN	The CANID is of type Extended (29 bits)	
	STANDARD_CAN	The CANID is of type Standard (11 bits)	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Network		

SWS Item	**CANIF599_Conf :**		
Name	CanIfRxPduDlc {CANIF_RXPDU_DLC}		
Description	Data Length code of received CAN L-PDUs used by the CAN Interface. Exa: DLC check. The data area size of a CAN L-PDU can have a range from 0 to 8 bytes.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 8		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME

	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Network		

SWS Item	**CANIF597_Conf :**		
Name	CanIfRxPduId {CANIF_RXPDUID}		
Description	ECU wide unique, symbolic handle for receive CAN L-PDU. The CanIfRxPduId is configurable at pre-compile and post-built time. It shall fulfill ANSI/AUTOSAR definitions for constant defines. Range: 0..max. number of defined CanRxPduIds		
Multiplicity	1		
Type	EcucIntegerParamDef (Symbolic Name generated for this parameter)		
Range	0 .. 4294967295		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

SWS Item	**CANIF600_Conf :**		
Name	CanIfRxPduReadData {CANIF_RXPDU_READDATA}		
Description	Enables and disables the Rx buffering for reading of received L-PDU data. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU dependency: CANIF_CANPDUID_READDATA_API must be enabled.		

SWS Item	**CANIF595_Conf :**		
Name	CanIfRxPduReadNotifyStatus {CANIF_RXPDU_READ_NOTIFYSTATUS}		
Description	Enables and disables receive indication for each receive CAN L-PDU for reading its notification status. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Module dependency: CANIF_READRXPDU_NOTIFY_STATUS_API must be enabled.		

SWS Item	**CANIF530_Conf :**
Name	CanIfRxPduUserRxIndicationName {CANIF_RXPDU_USERRXINDICATION_NAME}
Description	This parameter defines the name of the <User_RxIndication>. This parameter depends on the parameter CANIF_RXPDU_USERRXINDICATION_UL. If CANIF_RXPDU_USERRXINDICATION_UL equals CAN_TP, CAN_NM,

	PDUR, XCP or J1939TP, the name of the <User_RxIndication> is fixed. If CANIF_RXPDU_USERRXINDICATION_UL equals CDD, the name of the <User_RxIndication> is selectable.		
Multiplicity	0..1		
Type	EcucFunctionNameDef		
Default value	--		
maxLength	32		
minLength	1		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**CANIF529_Conf :**		
Name	CanIfRxPduUserRxIndicationUL {CANIF_RXPDU_USERRXINDICATION_UL}		
Description	This parameter defines the upper layer (UL) module to which the indication of the successfully received CANRXPDUID has to be routed via <User_RxIndication>. This <User_RxIndication> has to be invoked when the indication of the configured CANRXPDUID will be received by an Rx indication event from the CAN Driver module. If no upper layer (UL) module is configured, no <User_RxIndication> has to be called in case of an Rx indication event of the CANRXPDUID from the CAN Driver module.		
Multiplicity	0..1		
Type	EcucEnumerationParamDef		
Range	CAN_NM	CAN NM	
	CAN_TP	CAN TP	
	CDD	Complex Device Driver	
	J1939TP	J1939Tp	
	PDUR	PDU Router	
	XCP	Extended Calibration Protocol	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

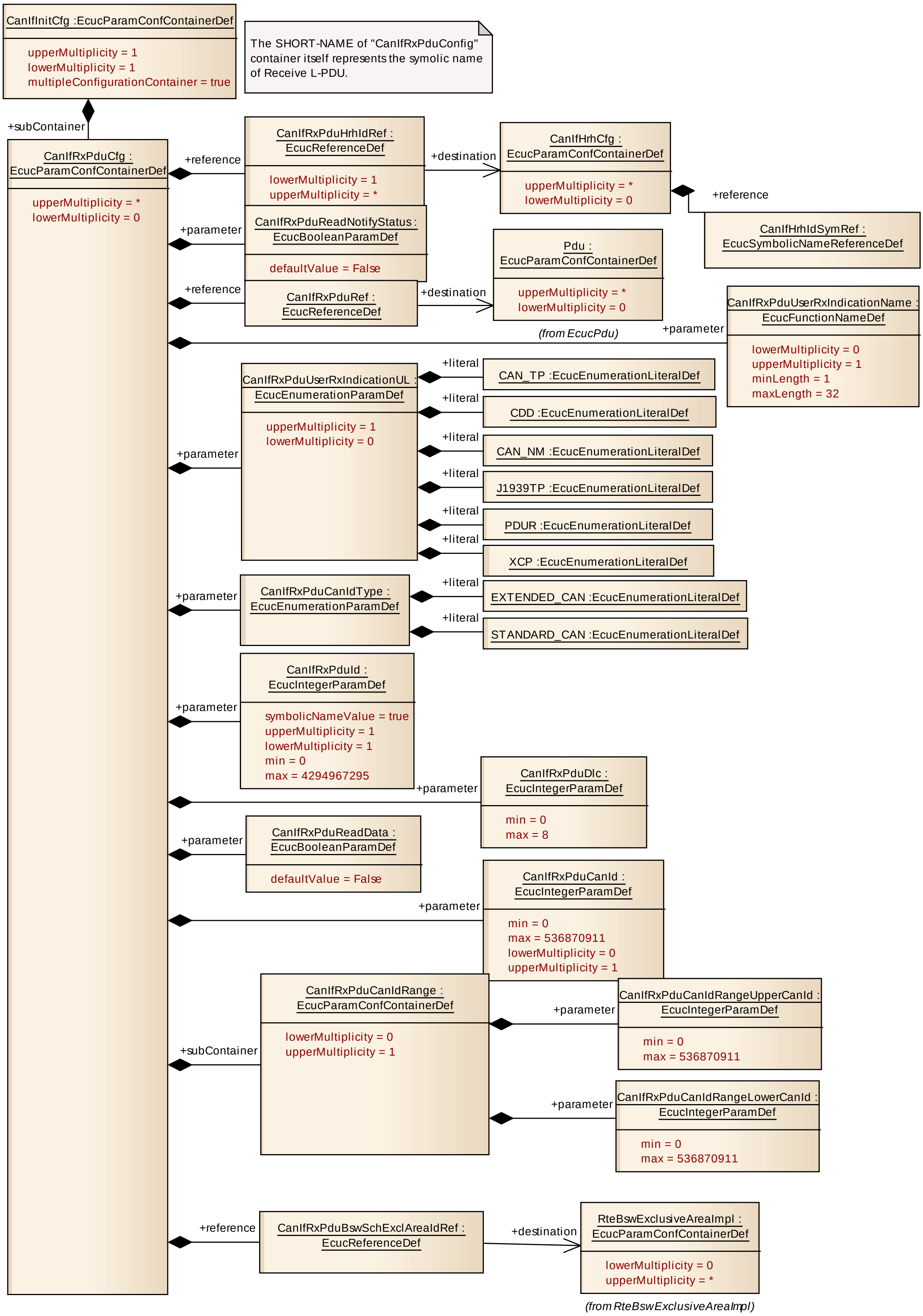
SWS Item	**CANIF669_Conf :**		
Name	CanIfRxPduBswSchExclAreaIdRef {CANIF_RXPDU_BSWSCH_EXCLAREAID_REF}		
Description	Reference to an exclusive area Id defined within the BSW Scheduler.		
Multiplicity	1		
Type	Reference to [RteBswExclusiveAreaImpl]		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**CANIF602_Conf :**
Name	CanIfRxPduHrhIdRef {CANIF_RXPDU_HRH_ID_REF}
Description	The HRH to which Rx L-PDU belongs to, is referred through this parameter.
Multiplicity	1..*
Type	Reference to [CanIfHrhCfg]

ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Module dependency: This information has to be derived from the CAN Driver configuration.		

SWS Item	**CANIF601_Conf :**		
Name	CanIfRxPduRef {CANIF_RXPDU_REF}		
Description	Reference to the "global" Pdu structure to allow harmonization of handle IDs in the COM-Stack.		
Multiplicity	1		
Type	Reference to [Pdu]		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanIfRxPduCanIdRange	0..1	Optional container that allows to map a range of CAN Ids to one PduId.
CanIfTTRxFrameTriggering	0..1	This container is only included and valid if TTCAN Interface SWS is used and TTCAN is enabled. Frame trigger for TTCAN reception. CanIfTTRxFrameTriggering is only included, if the controller supports TTCAN and a joblist is used for reception.


CanIfInitCfg :EcucParamConfContainerDef
upperMultiplicity = 1
lowerMultiplicity = 1
multipleConfigurationContainer = true
The SHORT-NAME of "CanIfRxPduConfig" container itself represents the symolic name of Receive L-PDU.
+subContainer
CanIfRxPduCfg : EcucParamConfContainerDef
upperMultiplicity = *
lowerMultiplicity = 0
+reference
CanIfRxPduHrhIdRef : EcucReferenceDef
lowerMultiplicity = 1
upperMultiplicity = *
+destination
CanIfHrhCfg : EcucParamConfContainerDef
upperMultiplicity = *
lowerMultiplicity = 0
+reference
CanIfHrhIdSymRef : EcucSymbolicNameReferenceDef
+parameter
CanIfRxPduReadNotifyStatus : EcucBooleanParamDef
defaultValue = False
Pdu : EcucParamConfContainerDef
upperMultiplicity = *
lowerMultiplicity = 0
(from EcucPdu)
+reference
CanIfRxPduRef : EcucReferenceDef
+destination
+parameter
CanIfRxPduUserRxIndicationName : EcucFunctionNameDef
lowerMultiplicity = 0
upperMultiplicity = 1
minLength = 1
maxLength = 32
+parameter
CanIfRxPduUserRxIndicationUL : EcucEnumerationParamDef
upperMultiplicity = 1
lowerMultiplicity = 0
+literal
CAN_TP :EcucEnumerationLiteralDef
+literal
CDD :EcucEnumerationLiteralDef
+literal
CAN_NM :EcucEnumerationLiteralDef
+literal
J1939TP :EcucEnumerationLiteralDef
+literal
PDUR :EcucEnumerationLiteralDef
+literal
XCP :EcucEnumerationLiteralDef
+parameter
CanIfRxPduCanIdType : EcucEnumerationParamDef
+literal
EXTENDED_CAN :EcucEnumerationLiteralDef
+literal
STANDARD_CAN :EcucEnumerationLiteralDef
+parameter
CanIfRxPduId : EcucIntegerParamDef
symbolicNameValue = true
upperMultiplicity = 1
lowerMultiplicity = 1
min = 0
max = 4294967295
+parameter
CanIfRxPduDlc : EcucIntegerParamDef
min = 0
max = 8
+parameter
CanIfRxPduReadData : EcucBooleanParamDef
defaultValue = False
+parameter
CanIfRxPduCanId : EcucIntegerParamDef
min = 0
max = 536870911
lowerMultiplicity = 0
upperMultiplicity = 1
+subContainer
CanIfRxPduCanIdRange : EcucParamConfContainerDef
lowerMultiplicity = 0
upperMultiplicity = 1
+parameter
CanIfRxPduCanIdRangeUpperCanId : EcucIntegerParamDef
min = 0
max = 536870911
+parameter
CanIfRxPduCanIdRangeLowerCanId : EcucIntegerParamDef
min = 0
max = 536870911
+reference
CanIfRxPduBswSchExclAreaIdRef : EcucReferenceDef
+destination
RteBswExclusiveAreaImpl : EcucParamConfContainerDef
lowerMultiplicity = 0
upperMultiplicity = *
(from RteBswExclusiveAreaImpl)

10.2.8 CanIfRxPduCanIdRange

SWS Item	**CANIF743_Conf :**
Container Name	CanIfRxPduCanIdRange
Description	Optional container that allows to map a range of CAN Ids to one PduId.
Configuration Parameters	

SWS Item	**CANIF745_Conf :**		
Name	CanIfRxPduCanIdRangeLowerCanId {CANIF_RX_PDU_CANID_RANGE_LOWER_CANID}		
Description	Lower CAN Identifier of a receive CAN L-PDU for identifier range definition, in which all CAN Ids are mapped to one PduId.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 536870911		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Module		

SWS Item	**CANIF744_Conf :**		
Name	CanIfRxPduCanIdRangeUpperCanId {CANIF_RX_PDU_CANID_RANGE_UPPER_CANID}		
Description	Upper CAN Identifier of a receive CAN L-PDU for identifier range definition, in which all CAN Ids are mapped to one PduId.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 536870911		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Module		

No Included Containers

10.2.9 CanIfDispatchCfg

SWS Item	**CANIF250_Conf :**
Container Name	CanIfDispatchCfg{CanInterfaceDispatcherConfiguration}
Description	Callback functions provided by upper layer modules of the CanIf. The callback functions defined in this container are common to all configured CAN Driver / CAN Transceiver Driver modules.
Configuration Parameters	

SWS Item	**CANIF791_Conf :**
Name	CanIfDispatchUserCheckTrcvWakeFlagIndicationName {CANIF_DISPATCH_USERCHECKTRCVWAKEFLAGINDICATION_NAME}
Description	This parameter defines the name of <User_ClearTrcvWufFlagIndication>. If CANIF_DISPATCH_USERCHECKTRCVWAKEFLAGINDICATION_UL equals

	CAN_SM the name of <User_CheckTrcvWakeFlagIndication> is fixed. If it equals CDD, the name is selectable. If CANIF_PUBLIC_PN_SUPPORT equals False, this parameter shall not be configurable.		
Multiplicity	0..1		
Type	EcucFunctionNameDef		
Default value	--		
maxLength	--		
minLength	--		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	dependency: CANIF_DISPATCH_USERCHECKTRCVWAKEFLAGINDICATION_UL, CANIF_PUBLIC_PN_SUPPORT		

SWS Item	**CANIF792_Conf :**		
Name	CanIfDispatchUserCheckTrcvWakeFlagIndicationUL {CANIF_DISPATCH_USERCHECKTRCVWAKEFLAGINDICATION_UL}		
Description	This parameter defines the upper layer module to which the CheckTrcvWakeFlagIndication from the Driver modules have to be routed. If CANIF_PUBLIC_PN_SUPPORT equals False, this parameter shall not be configurable.		
Multiplicity	0..1		
Type	EcucEnumerationParamDef		
Range	CAN_SM	CAN State Manager	
	CDD	Complex Device Driver	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	dependency: CANIF_PUBLIC_PN_SUPPORT		

SWS Item	**CANIF789_Conf :**		
Name	CanIfDispatchUserClearTrcvWufFlagIndicationName {CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_NAME}		
Description	This parameter defines the name of <User_ClearTrcvWufFlagIndication>. If CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_UL equals CAN_SM the name of <User_ClearTrcvWufFlagIndication> is fixed. If it equals CDD, the name is selectable. If CANIF_PUBLIC_PN_SUPPORT equals False, this parameter shall not be configurable.		
Multiplicity	0..1		
Type	EcucFunctionNameDef		
Default value	--		
maxLength	--		
minLength	--		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	dependency: CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_UL, CANIF_PUBLIC_PN_SUPPORT		

SWS Item	**CANIF790_Conf :**

Name	CanIfDispatchUserClearTrcvWufFlagIndicationUL {CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_UL}		
Description	This parameter defines the upper layer module to which the ClearTrcvWufFlagIndication from the Driver modules have to be routed. If CANIF_PUBLIC_PN_SUPPORT equals False, this parameter shall not be configurable.		
Multiplicity	0..1		
Type	EcucEnumerationParamDef		
Range	CAN_SM	CAN State Manager	
	CDD	Complex Device Driver	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	dependency: CANIF_PUBLIC_PN_SUPPORT		

SWS Item	**CANIF819_Conf :**		
Name	CanIfDispatchUserConfirmPnAvailabilityName {CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_NAME}		
Description	This parameter defines the name of <User_ConfirmPnAvailability>. If CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_UL equals CAN_SM the name of <User_ConfirmPnAvailability> is fixed. If it equals CDD, the name is selectable. If CANIF_PUBLIC_PN_SUPPORT equals False, this parameter shall not be configurable.		
Multiplicity	1		
Type	EcucFunctionNameDef		
Default value	--		
maxLength	--		
minLength	--		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	dependency: CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_UL, CANIF_PUBLIC_PN_SUPPORT		

SWS Item	**CANIF820_Conf :**		
Name	CanIfDispatchUserConfirmPnAvailabilityUL {CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_UL}		
Description	This parameter defines the upper layer module to which the ConfirmPnAvailability notification from the Driver modules have to be routed. If CANIF_PUBLIC_PN_SUPPORT equals False, this parameter shall not be configurable.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	CAN_SM	CAN State Manager	
	CDD	Complex Device Driver	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	dependency: CANIF_PUBLIC_PN_SUPPORT		

SWS Item	**CANIF525_Conf :**
Name	CanIfDispatchUserCtrlBusOffName {CANIF_DISPATCH_USERCTRLBUSOFF_NAME}

Description	This parameter defines the name of <User_ControllerBusOff>. This parameter depends on the parameter CANIF_USERCTRLBUSOFF_UL. If CANIF_USERCTRLBUSOFF_UL equals CAN_SM the name of <User_ControllerBusOff> is fixed. If CANIF_USERCTRLBUSOFF_UL equals CDD, the name of <User_ControllerBusOff> is selectable.		
Multiplicity	0..1		
Type	EcucFunctionNameDef		
Default value	--		
maxLength	32		
minLength	1		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: CANIF_DISPATCH_USERCTRLBUSOFF_UL		

SWS Item	**CANIF547_Conf :**		
Name	CanIfDispatchUserCtrlBusOffUL {CANIF_DISPATCH_USERCTRLBUSOFF_UL}		
Description	This parameter defines the upper layer (UL) module to which the notifications of all ControllerBusOff events from the CAN Driver modules have to be routed via <User_ControllerBusOff>. There is no possibility to configure no upper layer (UL) module as the provider of <User_ControllerBusOff>.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	CAN_SM	CAN State Manager	
	CDD	Complex Device Driver	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**CANIF683_Conf :**		
Name	CanIfDispatchUserCtrlModeIndicationName {CANIF_DISPATCH_USERCTRLMODEINDICATION_NAME}		
Description	This parameter defines the name of <User_ControllerModeIndication>. This parameter depends on the parameter CANIF_USERCTRLMODEINDICATION_UL. If CANIF_USERCTRLMODEINDICATION_UL equals CAN_SM the name of <User_ControllerModeIndication> is fixed. If CANIF_USERCTRLMODEINDICATION_UL equals CDD, the name of <User_ControllerModeIndication> is selectable.		
Multiplicity	0..1		
Type	EcucFunctionNameDef		
Default value	--		
maxLength	32		
minLength	1		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

	dependency: CANIF_DISPATCH_USERCTRLMODEINDICATION_UL

SWS Item	**CANIF684_Conf :**		
Name	CanIfDispatchUserCtrlModeIndicationUL {CANIF_DISPATCH_USERCTRLMODEINDICATION_UL}		
Description	This parameter defines the upper layer (UL) module to which the notifications of all ControllerTransition events from the CAN Driver modules have to be routed via <User_ControllerModeIndication>.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	CAN_SM	CAN State Manager	
	CDD	Complex Device Driver	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

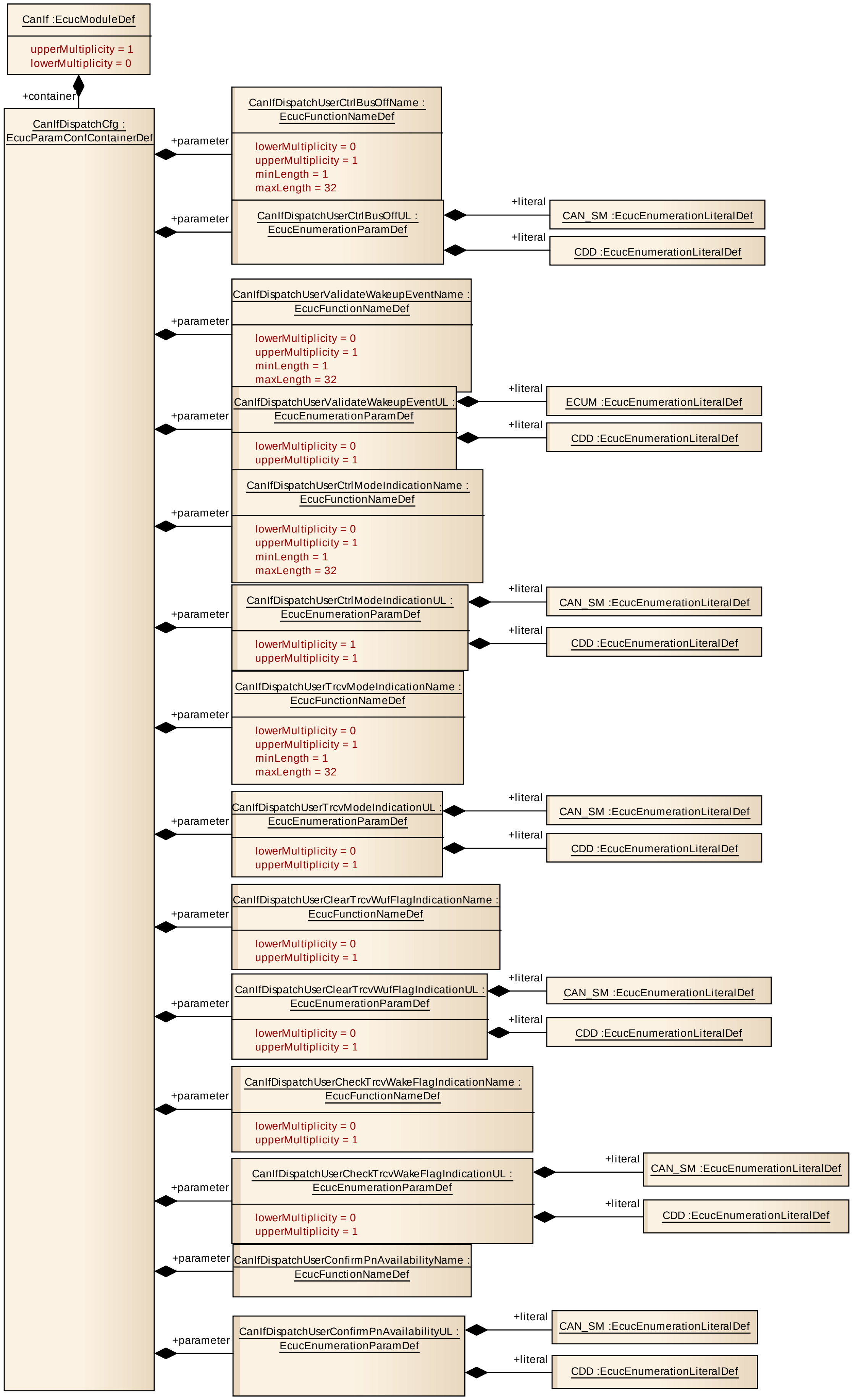
SWS Item	**CANIF685_Conf :**		
Name	CanIfDispatchUserTrcvModeIndicationName {CANIF_DISPATCH_USERTRCVMODEINDICATION_NAME}		
Description	This parameter defines the name of <User_TrcvModeIndication>. This parameter depends on the parameter CANIF_USERTRCVMODEINDICATION_UL. If CANIF_USERTRCVMODEINDICATION_UL equals CAN_SM the name of <User_TrcvModeIndication> is fixed. If CANIF_USERTRCVMODEINDICATION_UL equals CDD, the name of <User_TrcvModeIndication> is selectable.		
Multiplicity	0..1		
Type	EcucFunctionNameDef		
Default value	--		
maxLength	32		
minLength	1		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: CANIF_DISPATCH_USERTRCVMODEINDICATION_UL		

SWS Item	**CANIF686_Conf :**		
Name	CanIfDispatchUserTrcvModeIndicationUL {CANIF_DISPATCH_USERTRCVMODEINDICATION_UL}		
Description	This parameter defines the upper layer (UL) module to which the notifications of all TransceiverTransition events from the CAN Transceiver Driver modules have to be routed via <User_TrcvModeIndication>. If no UL module is configured, no upper layer callback function will be called.		
Multiplicity	0..1		
Type	EcucEnumerationParamDef		
Range	CAN_SM	CAN State Manager	
	CDD	Complex Device Driver	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**CANIF531_Conf :**		
Name	CanIfDispatchUserValidateWakeupEventName {CANIF_DISPATCH_USERVALIDATEWAKEUPEVENT_NAME}		
Description	This parameter defines the name of <User_ValidateWakeupEvent>. This parameter depends on the parameter CANIF_USERVALIDATEWAKEUPEVENT_UL. CANIF_USERVALIDATEWAKEUPEVENT_UL equals ECUM the name of <User_ValidateWakeupEvent> is fixed. CANIF_USERVALIDATEWAKEUPEVENT_UL equals CDD, the name of <User_ValidateWakeupEvent> is selectable. If parameter CANIF_WAKEUP_CHECK_VALIDATION_API is disabled, no <User_ValidateWakeupEvent> API can be configured.		
Multiplicity	0..1		
Type	EcucFunctionNameDef		
Default value	--		
maxLength	32		
minLength	1		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: CANIF_WAKEUP_CHECK_VALIDATION_API, CANIF_DISPATCH_USERVALIDATEWAKEUPEVENT_UL		

SWS Item	**CANIF549_Conf :**		
Name	CanIfDispatchUserValidateWakeupEventUL {CANIF_DISPATCH_USERVALIDATEWAKEUPEVENT_UL}		
Description	This parameter defines the upper layer (UL) module to which the notifications about positive former requested wake up sources have to be routed via <User_ValidateWakeupEvent>. If parameter CANIF_WAKEUP_CHECK_VALIDATION_API is disabled, this parameter cannot be configured.		
Multiplicity	0..1		
Type	EcucEnumerationParamDef		
Range	CDD	Complex Device Driver	
	ECUM	ECU State Manager	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: CANIF_WAKEUP_CHECK_VALIDATION_API		

No Included Containers


CanIf :EcucModuleDef
upperMultiplicity = 1
lowerMultiplicity = 0
+container
CanIfDispatchCfg :
EcucParamConfContainerDef
+parameter
CanIfDispatchUserCtrlBusOffName :
EcucFunctionNameDef
lowerMultiplicity = 0
upperMultiplicity = 1
minLength = 1
maxLength = 32
+parameter
CanIfDispatchUserCtrlBusOffUL :
EcucEnumerationParamDef
+literal
CAN_SM :EcucEnumerationLiteralDef
+literal
CDD :EcucEnumerationLiteralDef
+parameter
CanIfDispatchUserValidateWakeupEventName :
EcucFunctionNameDef
lowerMultiplicity = 0
upperMultiplicity = 1
minLength = 1
maxLength = 32
+parameter
CanIfDispatchUserValidateWakeupEventUL :
EcucEnumerationParamDef
lowerMultiplicity = 0
upperMultiplicity = 1
+literal
ECUM :EcucEnumerationLiteralDef
+literal
CDD :EcucEnumerationLiteralDef
+parameter
CanIfDispatchUserCtrlModeIndicationName :
EcucFunctionNameDef
lowerMultiplicity = 0
upperMultiplicity = 1
minLength = 1
maxLength = 32
+parameter
CanIfDispatchUserCtrlModeIndicationUL :
EcucEnumerationParamDef
lowerMultiplicity = 1
upperMultiplicity = 1
+literal
CAN_SM :EcucEnumerationLiteralDef
+literal
CDD :EcucEnumerationLiteralDef
+parameter
CanIfDispatchUserTrcvModeIndicationName :
EcucFunctionNameDef
lowerMultiplicity = 0
upperMultiplicity = 1
minLength = 1
maxLength = 32
+parameter
CanIfDispatchUserTrcvModeIndicationUL :
EcucEnumerationParamDef
lowerMultiplicity = 0
upperMultiplicity = 1
+literal
CAN_SM :EcucEnumerationLiteralDef
+literal
CDD :EcucEnumerationLiteralDef
+parameter
CanIfDispatchUserClearTrcvWufFlagIndicationName :
EcucFunctionNameDef
lowerMultiplicity = 0
upperMultiplicity = 1
+parameter
CanIfDispatchUserClearTrcvWufFlagIndicationUL :
EcucEnumerationParamDef
lowerMultiplicity = 0
upperMultiplicity = 1
+literal
CAN_SM :EcucEnumerationLiteralDef
+literal
CDD :EcucEnumerationLiteralDef
+parameter
CanIfDispatchUserCheckTrcvWakeFlagIndicationName :
EcucFunctionNameDef
lowerMultiplicity = 0
upperMultiplicity = 1
+parameter
CanIfDispatchUserCheckTrcvWakeFlagIndicationUL :
EcucEnumerationParamDef
lowerMultiplicity = 0
upperMultiplicity = 1
+literal
CAN_SM :EcucEnumerationLiteralDef
+literal
CDD :EcucEnumerationLiteralDef
+parameter
CanIfDispatchUserConfirmPnAvailabilityName :
EcucFunctionNameDef
+parameter
CanIfDispatchUserConfirmPnAvailabilityUL :
EcucEnumerationParamDef
+literal
CAN_SM :EcucEnumerationLiteralDef
+literal
CDD :EcucEnumerationLiteralDef

10.2.10 CanIfCtrlCfg

SWS Item	**CANIF546_Conf :**
Container Name	CanIfCtrlCfg{CanInterfaceControllerConfiguration}
Description	This container contains the configuration (parameters) of an adressed CAN controller by an underlying CAN Driver module. This container is configurable per CAN controller.
Configuration Parameters	

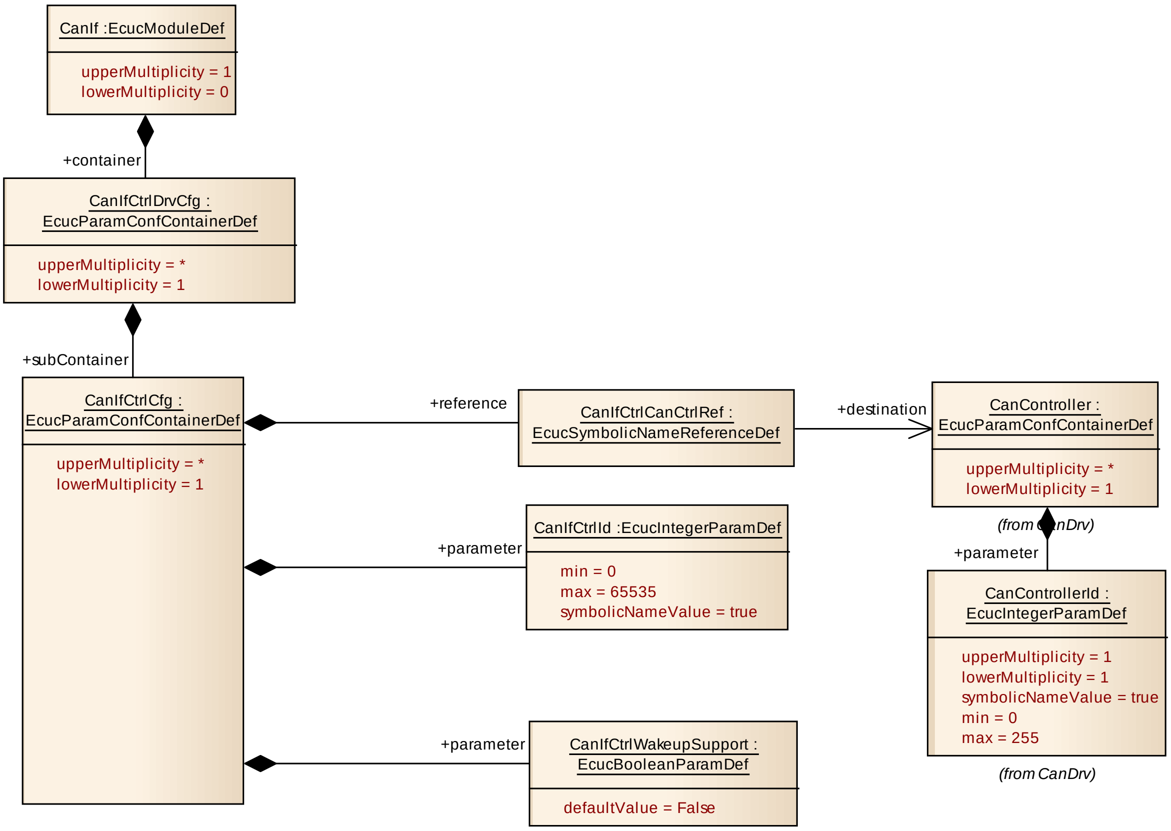
SWS Item	**CANIF647_Conf :**		
Name	CanIfCtrlId {CANIF_CTRL_ID}		
Description	This parameter abstracts from the CAN Driver specific parameter Controller. Each controller of all connected CAN Driver modules shall be assigned to one specific ControllerId of the CanIf. Range: 0..number of configured controllers of all CAN Driver modules		
Multiplicity	1		
Type	EcucIntegerParamDef (Symbolic Name generated for this parameter)		
Range	0 .. 65535		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: CAN Stack		

SWS Item	**CANIF637_Conf :**		
Name	CanIfCtrlWakeupSupport {CANIF_CTRL_WAKEUP_SUPPORT}		
Description	This parameter defines if a respective controller of the referenced CAN Driver modules is queriable for wake up events. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**CANIF636_Conf :**		
Name	CanIfCtrlCanCtrlRef {CANIF_CTRL_CAN_CONTROLLER_REF}		
Description	This parameter references to the logical handle of the underlying CAN controller from the CAN Driver module to be served by the CAN Interface module. The following parameters of CanController config container shall be referenced by this link: CanControllerId, CanWakeupSourceRef Range: 0..max. number of underlying supported CAN controllers		
Multiplicity	1		
Type	Reference to [CanController]		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE

	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: amount of CAN controllers		

No Included Containers



10.2.11 CanIfCtrlDrvCfg

SWS Item	**CANIF253_Conf :**
Container Name	CanIfCtrlDrvCfg{CanInterfaceControllerDriverConfiguration}
Description	Configuration parameters for all the underlying CAN Driver modules are aggregated under this container. For each CAN Driver module a seperate instance of this container has to be provided.
Configuration Parameters	

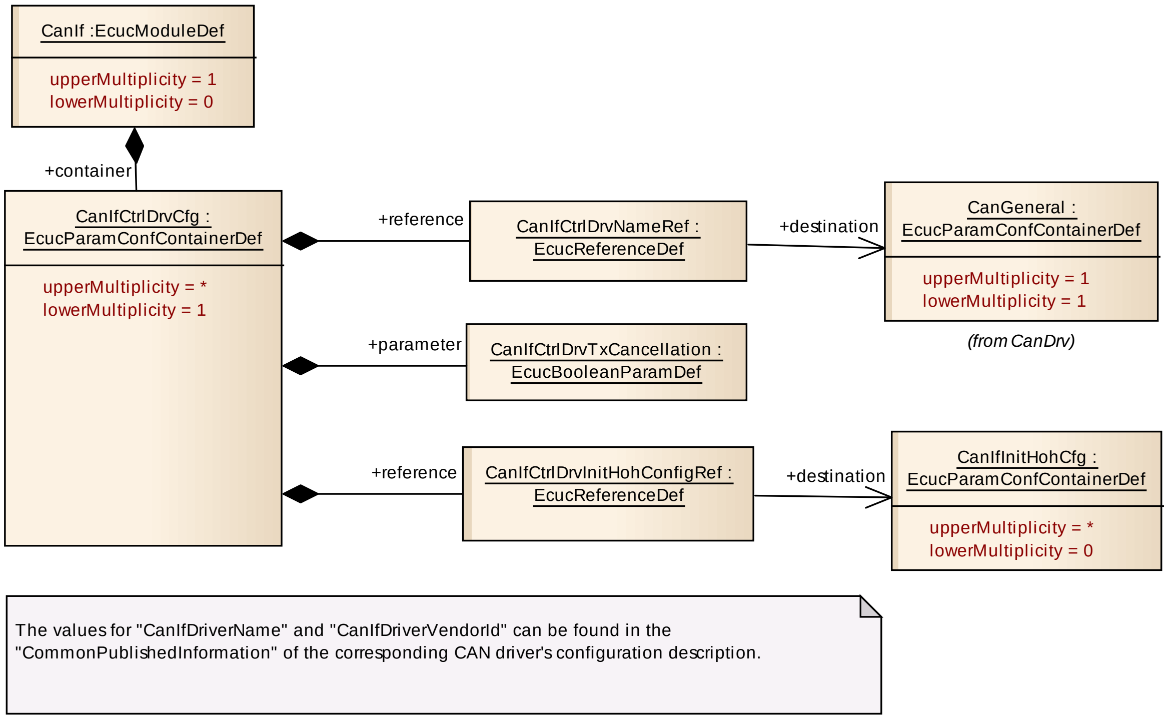
SWS Item	**CANIF640_Conf :**		
Name	CanIfCtrlDrvTxCancellation {CANIF_CTRLDRV_TX_CANCELLATION}		
Description	Selects whether transmit cancellation is supported and if the appropriate callback will be provided to the CAN Driver module. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	

	Post-build time	--	
Scope / Dependency	scope: Module dependency: CANIF_PUBLIC_TX_BUFFERING has to be enabled		

SWS Item	**CANIF642_Conf :**		
Name	CanIfCtrlDrvInitHohConfigRef {CANIF_CTRLDRV_INIT_HOH_CONFIG_REF}		
Description	Reference to the Init Hoh Configuration		
Multiplicity	1		
Type	Reference to [CanIfInitHohCfg]		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency			

SWS Item	**CANIF638_Conf :**		
Name	CanIfCtrlDrvNameRef {CANIF_CTRLDRV_NAME_REF}		
Description	CAN Interface Driver Reference. This reference can be used to get any information (Ex. Driver Name, Vendor ID) from the CAN driver. The CAN Driver name can be derived from the ShortName of the CAN driver module.		
Multiplicity	1		
Type	Reference to [CanGeneral]		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

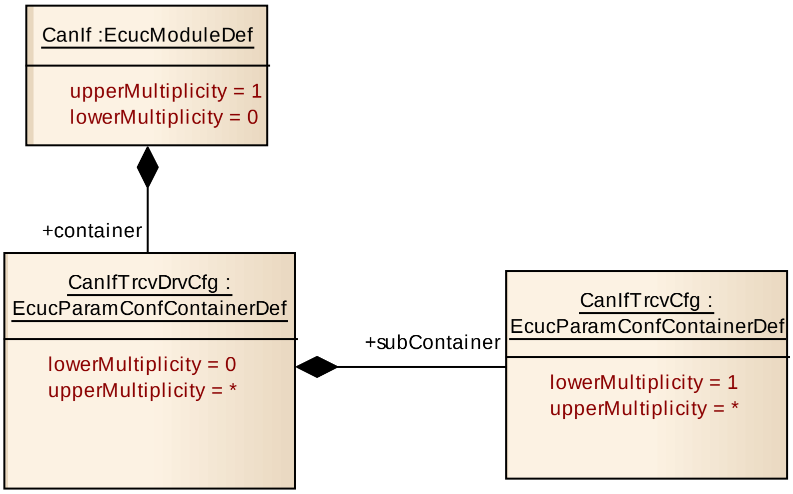
Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanIfCtrlCfg	1..*	This container contains the configuration (parameters) of an adressed CAN controller by an underlying CAN Driver module. This container is configurable per CAN controller.



10.2.12 CanIfTrcvDrvCfg

SWS Item	**CANIF273_Conf :**
Container Name	CanIfTrcvDrvCfg{CanInterfaceTransceiverDriverConfiguration}
Description	This container contains the configuration (parameters) of all addressed CAN transceivers by each underlying CAN Transceiver Driver module. For each CAN transceiver Driver a seperate instance of this container shall be provided.
Configuration Parameters	

Included Containers		
Container Name	*Multiplicity*	*Scope / Dependency*
CanIfTrcvCfg	1..*	This container contains the configuration (parameters) of one addressed CAN transceiver by the underlying CAN Transceiver Driver module. For each CAN transceiver a seperate instance of this container has to be provided.



10.2.13 CanIfTrcvCfg

SWS Item	**CANIF587_Conf :**
Container Name	CanIfTrcvCfg{CanInterfaceTransceiverConfiguration}
Description	This container contains the configuration (parameters) of one addressed CAN transceiver by the underlying CAN Transceiver Driver module. For each CAN transceiver a seperate instance of this container has to be provided.
Configuration Parameters	

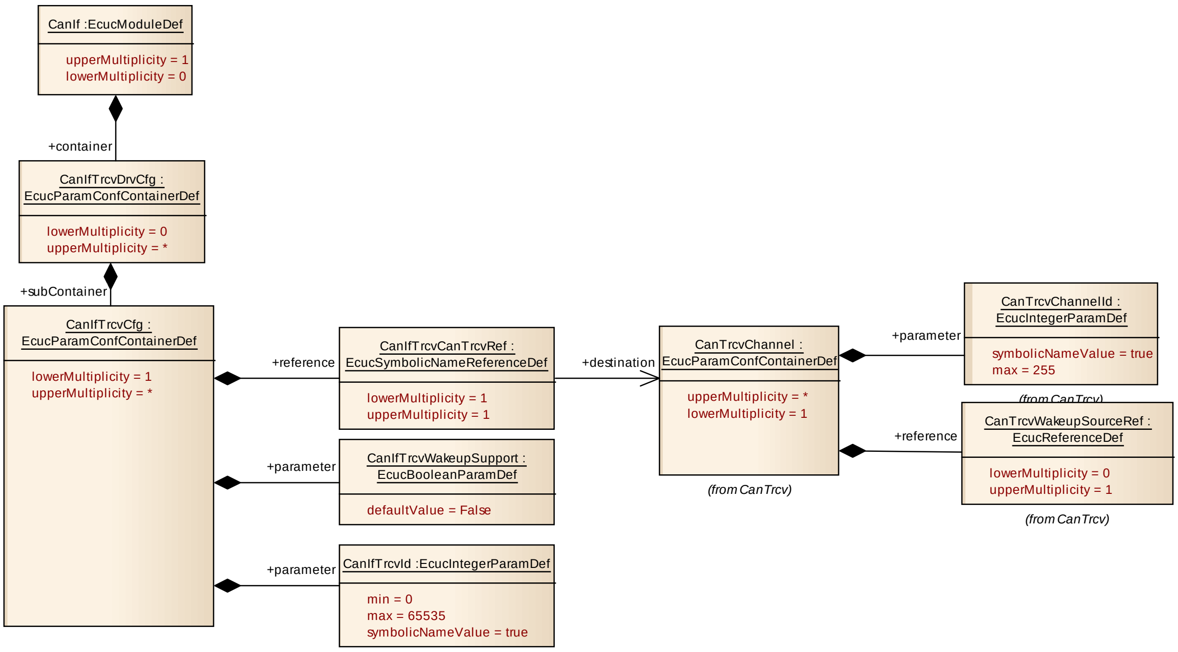
SWS Item	**CANIF654_Conf :**		
Name	CanIfTrcvId {CANIF_TRCV_ID}		
Description	This parameter abstracts from the CAN Transceiver Driver specific parameter Transceiver. Each transceiver of all connected CAN Transceiver Driver modules shall be assigned to one specific TransceiverId of the CanIf. Range: 0..number of configured transceivers of all CAN Transceiver Driver modules		
Multiplicity	1		
Type	EcucIntegerParamDef (Symbolic Name generated for this parameter)		
Range	0 .. 65535		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: CAN Stack		

SWS Item	**CANIF606_Conf :**
Name	CanIfTrcvWakeupSupport {CANIF_TRCV_WAKEUP_SUPPORT}
Description	This parameter defines if a respective transceiver of the referenced CAN Transceiver Driver modules is queriable for wake up events. True: Enabled False: Disabled
Multiplicity	1
Type	EcucBooleanParamDef

Default value	false		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**CANIF605_Conf :**		
Name	CanIfTrcvCanTrcvRef {CANIF_TRCV_CAN_TRANSCEIVER_REF}		
Description	This parameter references to the logical handle of the underlying CAN transceiver from the CAN transceiver driver module to be served by the CAN Interface module. Range: 0..max. number of underlying supported CAN transceivers		
Multiplicity	1		
Type	Reference to [CanTrcvChannel]		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: amount of CAN transceivers		

No Included Containers

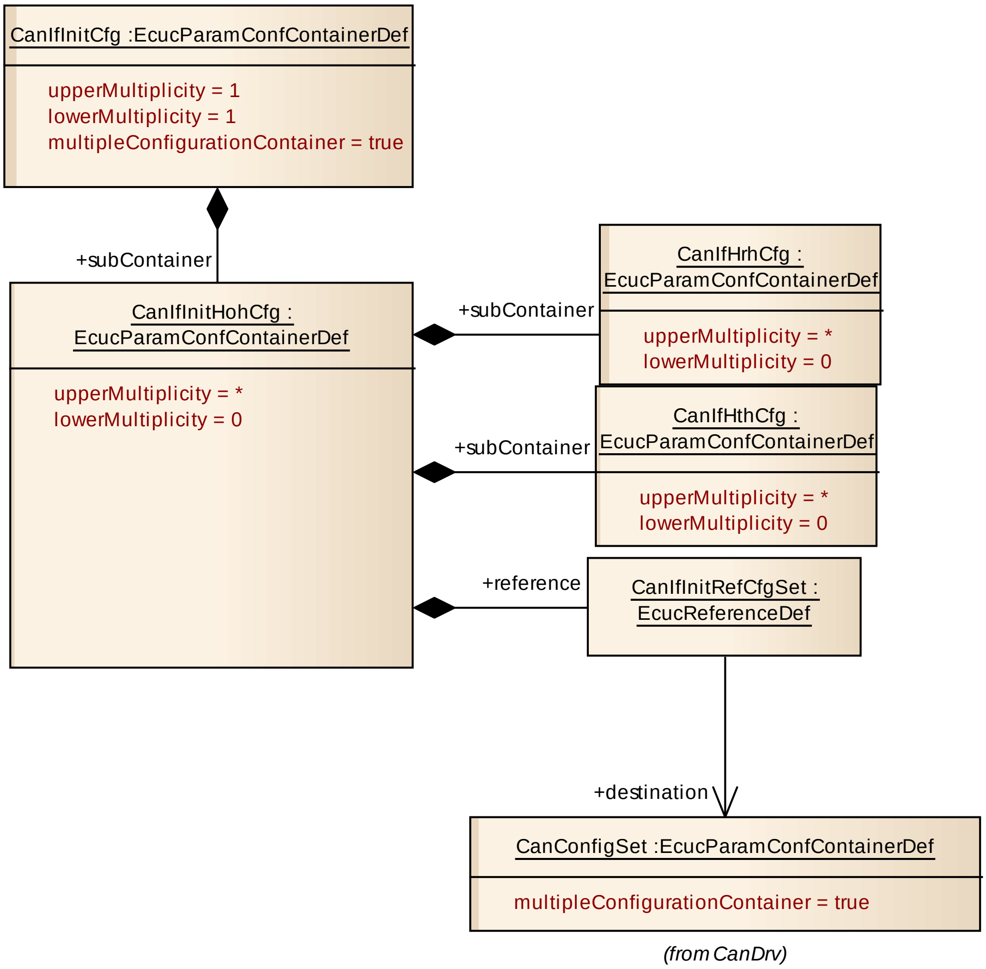


10.2.14 CanIfInitHohCfg

SWS Item	**CANIF257_Conf :**
Container Name	CanIfInitHohCfg{CANIF_INIT_HOH_CFG}
Description	This container contains the references to the configuration setup of each underlying CAN Driver.
Configuration Parameters	

SWS Item	**CANIF620_Conf :**		
Name	CanIfInitRefCfgSet {CANIF_INIT_REF_CFGSET}		
Description	Selects the CAN Interface specific configuration setup. This type of external data structure shall contain the post build initialization data for the CAN Interface for all underlying CAN Drivers.		
Multiplicity	1		
Type	Reference to [CanConfigSet]		
ConfigurationClass	*Pre-compile time*	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Module		

Included Containers		
Container Name	*Multiplicity*	*Scope / Dependency*
CanIfHrhCfg	0..*	This container contains configuration parameters for each hardware receive object (HRH).
CanIfHthCfg	0..*	This container contains parameters related to each HTH.


CanIfInitCfg :EcucParamConfContainerDef
upperMultiplicity = 1
lowerMultiplicity = 1
multipleConfigurationContainer = true
+subContainer
CanIfInitHohCfg : EcucParamConfContainerDef
upperMultiplicity = *
lowerMultiplicity = 0
+subContainer
CanIfHrhCfg : EcucParamConfContainerDef
upperMultiplicity = *
lowerMultiplicity = 0
+subContainer
CanIfHthCfg : EcucParamConfContainerDef
upperMultiplicity = *
lowerMultiplicity = 0
+reference
CanIfInitRefCfgSet : EcucReferenceDef
+destination
CanConfigSet :EcucParamConfContainerDef
multipleConfigurationContainer = true
(from CanDrv)

10.2.15 CanIfHthCfg

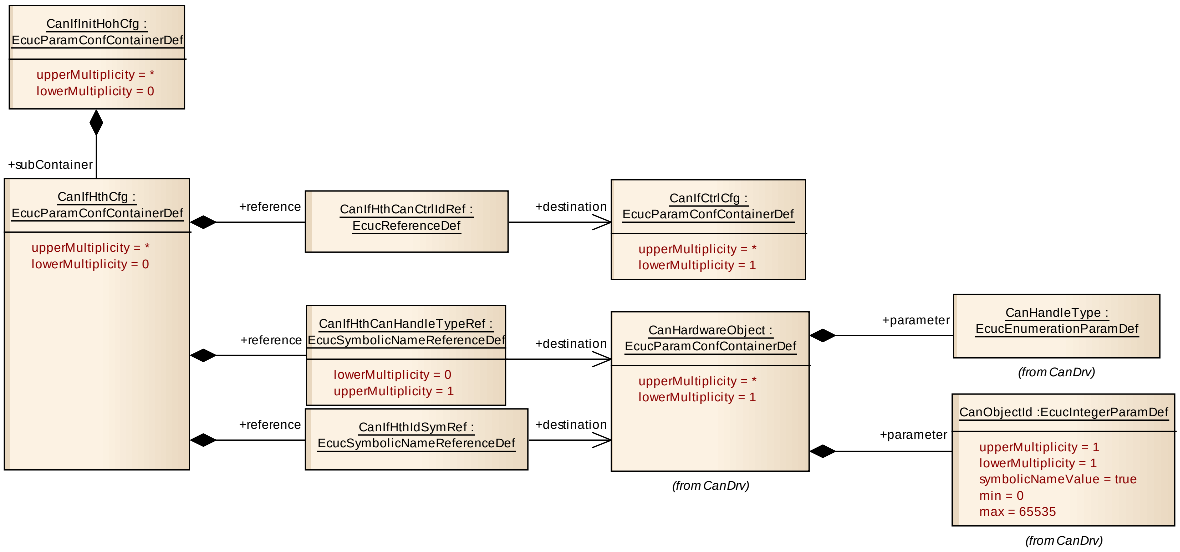
SWS Item	**CANIF258_Conf :**
Container Name	CanIfHthCfg{CanInterfaceHthConfiguration}
Description	This container contains parameters related to each HTH.
Configuration Parameters	

SWS Item	**CANIF625_Conf :**		
Name	CanIfHthCanCtrlIdRef {CANIF_HTH_CAN_CONTROLLER_ID_REF}		
Description	Reference to controller Id to which the HTH belongs to. A controller can contain one or more HTHs.		
Multiplicity	1		
Type	Reference to [CanIfCtrlCfg]		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency			

SWS Item	**CANIF626_Conf :**		
Name	CanIfHthCanHandleTypeRef {CANIF_HTH_HANDLETYPE_REF}		
Description	The parameter refers to a particular HTH object in the CAN Driver Module configuration. The type of the HTH can either be Full-CAN or Basic-CAN. The type of HTHs is defined in the CAN Driver Module and hence it is derived from CAN Driver Configuration of a Hardware Object. Please note that this reference is deprecated and is kept only for backward compatibility reasons. CanIfHthIdSymRef shall be used instead to get the CanHandleType and CanObjectId of CAN Driver. In the next major release this reference will be deleted.		
Multiplicity	0..1		
Type	Reference to [CanHardwareObject]		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency			

SWS Item	**CANIF627_Conf :**		
Name	CanIfHthIdSymRef {CANIF_HTH_ID_SYMREF}		
Description	The parameter refers to a particular HTH object in the CanDrv configuration (see CanHardwareObject CAN324_Conf). The CanIf receives the following information of the CanDrv module by this reference: - CanHandleType (see CAN323_Conf) - CanObjectId (see CAN326_Conf)		
Multiplicity	1		
Type	Reference to [CanHardwareObject]		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency			

No Included Containers



10.2.16 CanIfHrhCfg

SWS Item	**CANIF259_Conf :**
Container Name	CanIfHrhCfg{CanInterfaceHrhConfiguration}
Description	This container contains configuration parameters for each hardware receive object (HRH).
Configuration Parameters	

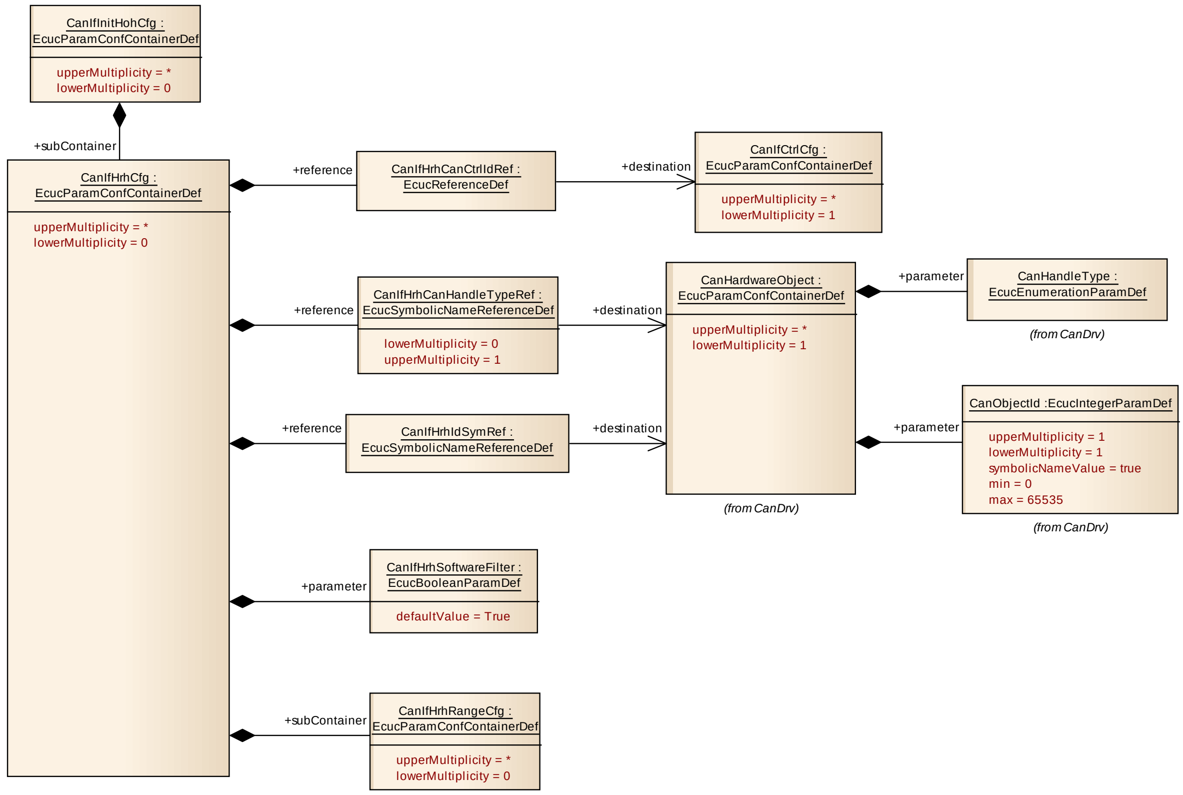
SWS Item	**CANIF632_Conf :**		
Name	CanIfHrhSoftwareFilter {CANIF_HRH_SOFTWARE_FILTER}		
Description	Selects the hardware receive objects by using the HRH range/list from CAN Driver configuration to define, for which HRH a software filtering has to be performed at during receive processing. True: Software filtering is enabled False: Software filtering is enabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	true		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: Module		

SWS Item	**CANIF631_Conf :**		
Name	CanIfHrhCanCtrlIdRef {CANIF_HRH_CAN_CTRL_ID_REF}		
Description	Reference to controller Id to which the HRH belongs to. A controller can contain one or more HRHs.		
Multiplicity	1		
Type	Reference to [CanIfCtrlCfg]		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency			

SWS Item	**CANIF633_Conf :**		
Name	CanIfHrhCanHandleTypeRef {CANIF_HRH_HANDLETYPE_REF}		
Description	The parameter refers to a particular HRH object in the CAN Driver Module configuration. The type of the HRH can either be Full-CAN or Basic-CAN. The type of HRHs is defined in the CAN Driver Module and hence it is derived from CAN Driver Configuration of a Hardware Object. If BasicCAN is configured, software filtering is enabled. Please note that this reference is deprecated and is kept only for backward compatibility reasons. CanIfHthIdSymRef shall be used instead to get the CanHandleType and CanObjectId of CAN Driver. In the next major release this reference will be deleted.		
Multiplicity	0..1		
Type	Reference to [CanHardwareObject]		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency			

SWS Item	**CANIF634_Conf :**		
Name	CanIfHrhIdSymRef {CANIF_HRH_ID_SYMREF}		
Description	The parameter refers to a particular HRH object in the CanDrv configuration (see CanHardwareObject CAN324_Conf). The CanIf receives the following information of the CanDrv module by this reference: - CanHandleType (see CAN323_Conf) - CanObjectId (see CAN326_Conf)		
Multiplicity	1		
Type	Reference to [CanHardwareObject]		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency			

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanIfHrhRangeCfg	0..*	Defines the parameters required for configurating multiple CANID ranges for a given same HRH.



10.2.17 CanIfHrhRangeCfg

SWS Item	**CANIF628_Conf :**
Container Name	CanIfHrhRangeCfg{CanInterfaceHrhRangeConfiguration}
Description	Defines the parameters required for configurating multiple CANID ranges for a given same HRH.
Configuration Parameters	

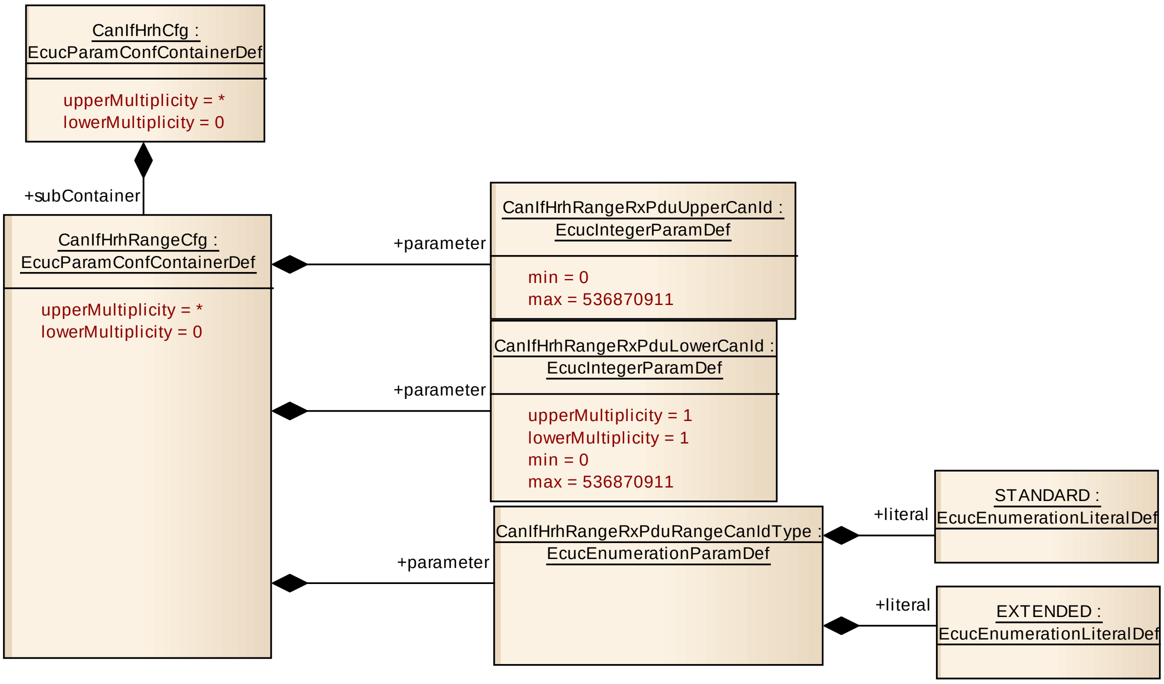
SWS Item	**CANIF629_Conf :**		
Name	CanIfHrhRangeRxPduLowerCanId {CANIF_HRHRANGE_LOWER_CANID}		
Description	Lower CAN Identifier of a receive CAN L-PDU for identifier range definition, in which all CAN Ids shall pass the software filtering.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 536870911		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Module		

SWS Item	**CANIF644_Conf :**
Name	CanIfHrhRangeRxPduRangeCanIdType {CANIF_HRHRANGE_CANIDTYPE}
Description	Specifies whether a configured Range of CAN Ids shall only consider standard CAN Ids or extended CAN Ids.

Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	EXTENDED	All the CANIDs are of type extended only (29 bit).	
	STANDARD	All the CANIDs are of type standard only (11bit).	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Module		

SWS Item	**CANIF630_Conf :**		
Name	CanIfHrhRangeRxPduUpperCanId {CANIF_HRHRANGE_UPPER_CANID}		
Description	Upper CAN Identifier of a receive CAN L-PDU for identifier range definition, in which all CAN Ids shall pass the software filtering.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 536870911		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Module		

No Included Containers



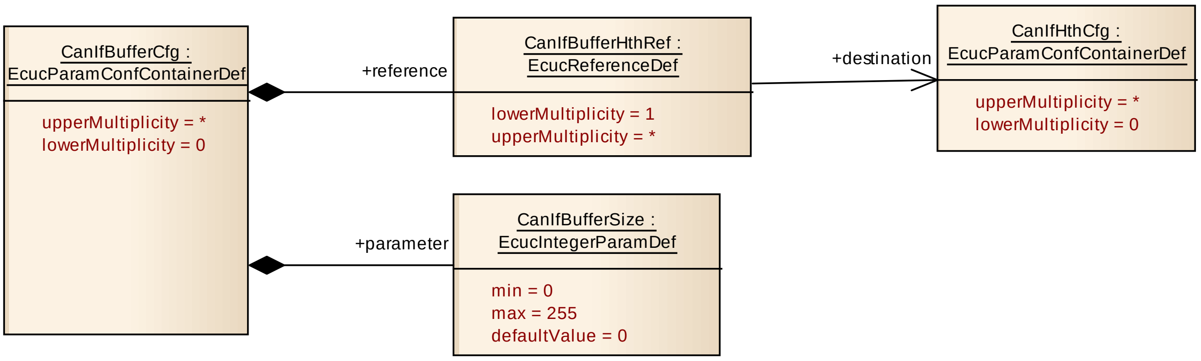
10.2.18 CanIfBufferCfg

SWS Item	**CANIF832_Conf :**
Container Name	CanIfBufferCfg{CANIF_BUFFER_CFG}
Description	This container contains the Txbuffer configuration. Multiple buffers with different sizes could be configured. If CanIfBufferSize (CANIF834_Conf) equals 0, the CanIf Tx L-PDU only refers via this CanIfBufferCfg the corresponding CanIfHthCfg.
Configuration Parameters	

SWS Item	**CANIF834_Conf :**		
Name	CanIfBufferSize {CANIF_BUFFER_SIZE}		
Description	This parameter defines the number of CanIf Tx L-PDUs which can be buffered in one Txbuffer. If this value equals 0, the CanIf does not perform Txbuffering for the CanIf Tx L-PDUs which are assigned to this Txbuffer. If CanIfPublicTxBuffering equals False, this parameter equals 0 for all TxBuffer. If the CanHandleType of the referred HTH equals FULL, this parameter equals 0 for this TxBuffer.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 255		
Default value	0		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local dependency: CanIfPublicTxBuffering, CanHandleType		

SWS Item	**CANIF833_Conf :**		
Name	CanIfBufferHthRef {CANIF_BUFFER_HTH_REF}		
Description	Reference to HTH, that defines the hardware object or the pool of hardware objects configured for transmission. All the CanIf Tx L-PDUs refer via the CanIfBufferCfg and this parameter to the HTHs if TxBuffering is enabled, or not. Each HTH shall not be assigned to more than one buffer.		
Multiplicity	1..*		
Type	Reference to [CanIfHthCfg]		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

No Included Containers


CanIfBufferCfg :
EcucParamConfContainerDef
upperMultiplicity = *
lowerMultiplicity = 0
+reference
CanIfBufferHthRef :
EcucReferenceDef
lowerMultiplicity = 1
upperMultiplicity = *
+destination
CanIfHthCfg :
EcucParamConfContainerDef
upperMultiplicity = *
lowerMultiplicity = 0
+parameter
CanIfBufferSize :
EcucIntegerParamDef
min = 0
max = 255
defaultValue = 0

11 Changes to release 4.0.3

11.1 Deleted SWS items

SWS Item	***Rationale***
CANIF013	
CANIF024	Removal of CanIfCtrlDrvRxIndication and CanIfCtrlDrvTxConfirmation configuraion parameters
CANIF114	Improvement of transmit buffer handling
CANIF295	
CANIF309	Centralized UnInit specification item (CANIF156)
CANIF314	Centralized UnInit specification item (CANIF156)
CANIF327	Centralized UnInit specification item (CANIF156)
CANIF332	Centralized UnInit specification item (CANIF156)
CANIF337	Centralized UnInit specification item (CANIF156)
CANIF342	Centralized UnInit specification item (CANIF156)
CANIF347	Centralized UnInit specification item (CANIF156)
CANIF354	Centralized UnInit specification item (CANIF156)
CANIF359	Centralized UnInit specification item (CANIF156)
CANIF365	Centralized UnInit specification item (CANIF156)
CANIF369	Centralized UnInit specification item (CANIF156)
CANIF396	
CANIF399	Centralized UnInit specification item (CANIF156)
CANIF403	
CANIF405	Centralized UnInit specification item (CANIF156)
CANIF420	Centralized UnInit specification item (CANIF156)
CANIF425	Centralized UnInit specification item (CANIF156)
CANIF430	Centralized UnInit specification item (CANIF156)
CANIF441	Centralized UnInit specification item (CANIF156)
CANIF452	
CANIF453	
CANIF458	
CANIF459	
CANIF484	CANIF484: pending transmit requests ?
CANIF534	Centralized UnInit specification item (CANIF156)
CANIF561	
CANIF562	
CANIF639_Conf	Removal of CanIfCtrlDrvRxIndication and CanIfCtrlDrvTxConfirmation configuraion parameters
CANIF641_Conf	Removal of CanIfCtrlDrvRxIndication and CanIfCtrlDrvTxConfirmation configuraion parameters
CANIF676	
CANIF680	
CANIF721	
CANIF722	
CANIF701	Centralized UnInit specification item (CANIF156)
CANIF707	Centralized UnInit specification item (CANIF156)
CANIF735	Centralized UnInit specification item (CANIF156)

11.2 Replaced SWS items

SWS Item of Release 2	replaced by SWS Item	Rationale

11.3 Changed SWS items

SWS Item	Rationale
CANIF003	Changed service to asynchronous
CANIF054	Improvement of transmit buffer handling
CANIF063	Improvement of transmit buffer handling
CANIF068	Improvement of transmit buffer handling
CANIF118	CANIF page 66-67:contradiction between CANIF073, CANIF489 and CANIF118
CANIF168	Changed CANIF_E_INVALID_DLC from production to development error
CANIF154	Changed CANIF_E_STOPPED from production to development error; changed CANIF_E_SLEEP and CANIF_INVALID_DLC from production to development error
CANIF179	
CANIF226	
CANIF286	
CANIF287	Changed service to asynchronous
CANIF297	Clarification/Improvment on DLC Check description
CANIF381	Improvement of transmit buffer handling
CANIF382	
CANIF414	Removal of CanIfCtrlDrvRxIndication and CanIfCtrlDrvTxConfirmation configuraion parameters
CANIF423	Removal of CanIfCtrlDrvRxIndication and CanIfCtrlDrvTxConfirmation configuraion parameters
CANIF466	Improvement of transmit buffer handling
CANIF468	Redundant information in CanIfHthCfg container
CANIF520	Changed service to Synchronous
CANIF626_Conf	Redundant information in CanIfHthCfg container
CANIF627_Conf	Redundant information in CanIfHthCfg container
CANIF633_Conf	Redundant information in CanIfHthCfg container
CANIF634_Conf	Redundant information in CanIfHthCfg container
CANIF679	Changed CANIF_E_SLEEP from production to development error
CANIF723	

11.4 Added SWS items

SWS Item	Rationale
CANIF747	Network Management Extensions for Partial Networking
CANIF748	Network Management Extensions for Partial Networking
CANIF749	Network Management Extensions for Partial Networking
CANIF750	Network Management Extensions for Partial Networking

CANIF751	Network Management Extensions for Partial Networking
CANIF752	Network Management Extensions for Partial Networking
CANIF757	[CanSm] Instruction order of Entering NoCom
CANIF758	CDD support of CanIf_TrcvModeIndication
CANIF759	[CanSm] Instruction order of Entering NoCom
CANIF760	[CanSm] Instruction order of Entering NoCom
CANIF761	[CanSm] Instruction order of Entering NoCom
CANIF762	[CanSm] Instruction order of Entering NoCom
CANIF763	[CanSm] Instruction order of Entering NoCom
CANIF764	Changed from CANIF705 to this item, to be consistent to release 3.2
CANIF765	[CanSm] Instruction order of Entering NoCom
CANIF766	[CanSm] Instruction order of Entering NoCom
CANIF770	[CanSm] Instruction order of Entering NoCom
CANIF771	[CanSm] Instruction order of Entering NoCom
CANIF772_Conf	Network Management Extensions for Partial Networking
CANIF773_Conf	Network Management Extensions for Partial Networking
CANIF774	
CANIF775	Change of baudrate within UDS service linkcontrol
CANIF776	Change of baudrate within UDS service linkcontrol
CANIF778	Change of baudrate within UDS service linkcontrol
CANIF779	Change of baudrate within UDS service linkcontrol
CANIF780	Change of baudrate within UDS service linkcontrol
CANIF782	Change of baudrate within UDS service linkcontrol
CANIF783	Change of baudrate within UDS service linkcontrol
CANIF784	Change of baudrate within UDS service linkcontrol
CANIF785	Change of baudrate within UDS service linkcontrol
CANIF786	Change of baudrate within UDS service linkcontrol
CANIF787	Change of baudrate within UDS service linkcontrol
CANIF788	[CanSm] Instruction order of Entering NoCom
CANIF789_Conf	[CanSm] Instruction order of Entering NoCom
CANIF790_Conf	[CanSm] Instruction order of Entering NoCom
CANIF791_Conf	[CanSm] Instruction order of Entering NoCom
CANIF792_Conf	[CanSm] Instruction order of Entering NoCom
CANIF793	[CanSm] Instruction order of Entering NoCom
CANIF794	[CanSm] Instruction order of Entering NoCom
CANIF795	[CanSm] Instruction order of Entering NoCom
CANIF796	[CanSm] Instruction order of Entering NoCom
CANIF797	[CanSm] Instruction order of Entering NoCom
CANIF798	[CanSm] Instruction order of Entering NoCom
CANIF799	[CanSm] Instruction order of Entering NoCom
CANIF800	[CanSm] Instruction order of Entering NoCom
CANIF801	[CanSm] Instruction order of Entering NoCom
CANIF802	[CanSm] Instruction order of Entering NoCom
CANIF803	[CanSm] Instruction order of Entering NoCom
CANIF804	[CanSm] Instruction order of Entering NoCom
CANIF805	[CanSm] Instruction order of Entering NoCom
CANIF806	[CanSm] Instruction order of Entering NoCom
CANIF807	[CanSm] Instruction order of Entering NoCom
CANIF808	[CanSm] Instruction order of Entering NoCom
CANIF809	[CanSm] Instruction order of Entering NoCom
CANIF810	[CanSm] Instruction order of Entering NoCom
CANIF811	[CanSm] Instruction order of Entering NoCom
CANIF812	[CanSm] Instruction order of Entering NoCom
CANIF813	[CanSm] Instruction order of Entering NoCom
CANIF814	[CanSm] Instruction order of Entering NoCom
CANIF815	Handling if PN functionality is disabled in the Trcv
CANIF816	Handling if PN functionality is disabled in the Trcv
CANIF817	Handling if PN functionality is disabled in the Trcv

CANIF818	Handling if PN functionality is disabled in the Trcv
CANIF819_Conf	Handling if PN functionality is disabled in the Trcv
CANIF820_Conf	Handling if PN functionality is disabled in the Trcv
CANIF821	Handling if PN functionality is disabled in the Trcv
CANIF822	Handling if PN functionality is disabled in the Trcv
CANIF823	Handling if PN functionality is disabled in the Trcv
CANIF824	Handling if PN functionality is disabled in the Trcv
CANIF825	Handling if PN functionality is disabled in the Trcv
CANIF826	Handling if PN functionality is disabled in the Trcv
CANIF827	Handling if PN functionality is disabled in the Trcv
CANIF828	Incoherence for the returned error in the service CanIf_CancelTxConfirmation()
CANIF829	Clarification/Improvment on DLC Check description
CANIF830	Clarification/Improvment on DLC Check description
CANIF831_Conf	Improvement of transmit buffer handling
CANIF832_Conf	Improvement of transmit buffer handling
CANIF833_Conf	Improvement of transmit buffer handling
CANIF834_Conf	Improvement of transmit buffer handling
CANIF835	Improvement of transmit buffer handling
CANIF836	Improvement of transmit buffer handling
CANIF837	Improvement of transmit buffer handling

12 Not applicable requirements

[CANIF999]「These requirements are not applicable to this specification.」(BSW159, BSW167, BSW170, BSW00416, BSW168, BSW00423, BSW00424, BSW00425, BSW00426, BSW00427, BSW00428, BSW00429, BSW00431, BSW00432, BSW00433, BSW00434, BSW00336, BSW00417, BSW164, BSW00326, BSW007, BSW00307, BSW00373, BSW00435, BSW00328, BSW00378, BSW00306, BSW00308, BSW00309, BSW00376, BSW00330, BSW172, BSW010, BSW00341, BSW00334, BSW01139, BSW01014, BSW01024)