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

Document Version	6.2.0
Document Status	Final
Part of Release	4.1
Revision	3

Document Change History

Date	Version	Changed by	Description
31.03.2014	6.2.0	AUTOSAR Release Management	• Removed BSW Exclusive areas • Set ICOM support to optional • `Can_IdType` handling • Small improvements
10.10.2013	6.1.0	AUTOSAR Release Management	• Restricted PDU mode changes • Removed critical section handling description in chapter 9 • Set CanIfInitRefCfgSet oboslete • Pretended Networking section • Small improvements
22.02.2013	6.0.0	AUTOSAR Administration	• CAN FD (without DLC extension) • Pretended Networking (ICOM) • Heavy Duty Vehicle (J1939) support • PduModes and PnTxFilter for clean wake-up • Relation between PDUs & HOHs • Post-build loadable concept

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

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

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.

As depicted in Figure 1.1 the CAN Interface module is located between the low level CAN device drivers (CAN Driver [1] and Transceiver Driver [2]) and the upper communication service layers (i.e. CAN State Manager [3], CAN Network Management [4], CAN Transport Protocol [5], PDU Router [6]). 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 Managers module based on a physical CAN channel related view.

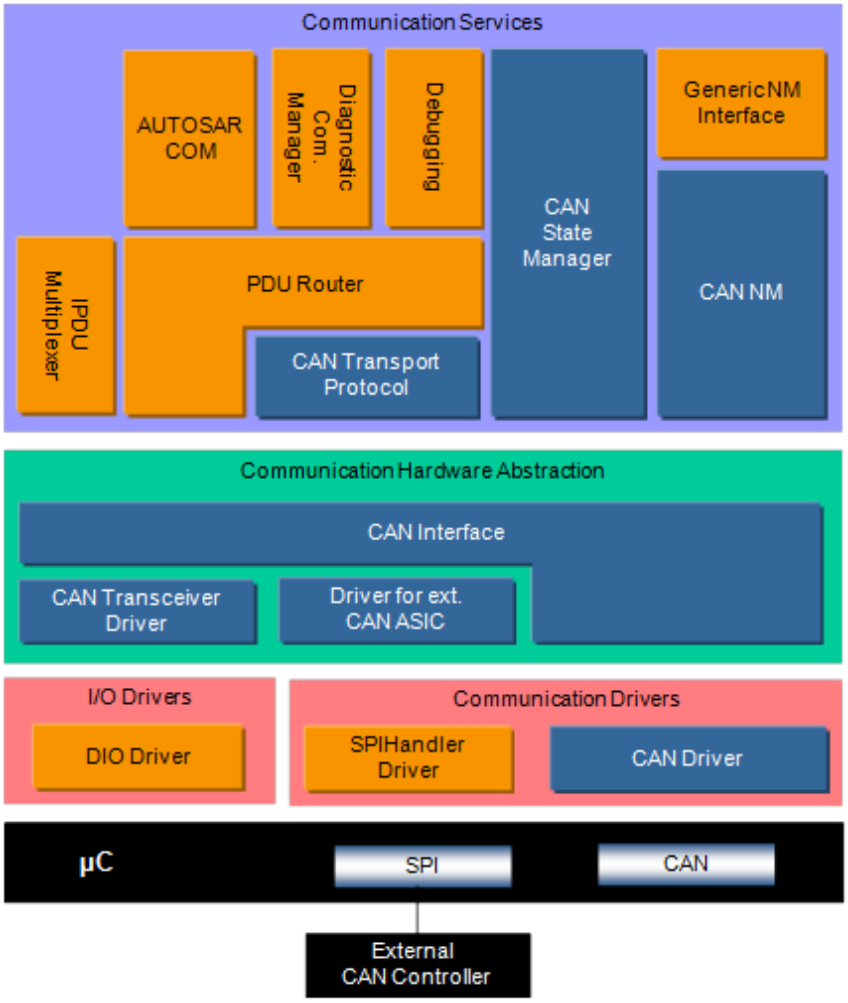

Figure 1.1: AUTOSAR CAN Layer Model (see [7])

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.

CanIf 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 is based on CAN L-SDUs, whereas APIs for control and mode handling provides a CAN Controller related view.

In case of Transmit Requests CanIf completes the L-PDU transmission with corresponding parameters and relays the CAN L-PDU via the appropriate CanDrv to the CAN Controller. At reception CanIf distributes the Received L-PDUs as L-SDUs to the upper layer. The assignment between Receive L-SDU and upper layer is statically configured. At transmit confirmation CanIf 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 [8, AUTOSAR glossary].

Abbreviation / Acronym:	Description:
CAN L-PDU	CAN Protocol Data Unit. Consists of an identifier, DLC and data (SDU). Visible to the CAN driver.
CAN L-SDU	CAN Service Data Unit. Data that are transported inside the CAN L-PDU. Visible to the upper layers of the CAN interface (e.g. PDU Router).
CanDrv	CAN Driver module
CanId	CAN Identifier
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)
ComM	Communication Manager module
DCM	Diagnostic Communication Manager module
EcuM	ECU State Manager module
HOH	CAN hardware object handle
HRH	CAN hardware receive handle
HTH	CAN hardware transmit handle
J1939Nm	J1939 Network Management module
J1939Tp	J1939 Transport Layer module
PduR	PDU Router module
PN	Partial Networking
SchM	Scheduler Module

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

Terms:	Description:
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 upper layer 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.

Terms:	Description:
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 & related standards and norms

[1] Specification of CAN Driver
AUTOSAR_SWS_CANDriver

[2] Specification of CAN Transceiver Driver
AUTOSAR_SWS_CANTransceiverDriver

[3] Specification of CAN State Manager
AUTOSAR_SWS_CANStateManager

[4] Specification of CAN Network Management
AUTOSAR_SWS_CANNetworkManagement

[5] Specification of CAN Transport Layer
AUTOSAR_SWS_CANTransportLayer

[6] Specification of PDU Router
AUTOSAR_SWS_PDURouter

[7] Layered Software Architecture
AUTOSAR_EXP_LayeredSoftwareArchitecture

[8] Glossary
AUTOSAR_TR_Glossary

[9] General Specification of Basic Software Modules
AUTOSAR_SWS_BSWGeneral

[10] General Requirements on Basic Software Modules
AUTOSAR_SRS_BSWGeneral

[11] Requirements on CAN
AUTOSAR_SRS_CAN

[12] Specification of Standard Types
AUTOSAR_SWS_StandardTypes

[13] ISO 11898-1:2003 - Road vehicles – Controller area network (CAN)

[14] Specification of ECU State Manager
AUTOSAR_SWS_ECUStateManager

[15] Specification of ECU Configuration
AUTOSAR_TPS_ECUConfiguration

3.2 Related specification

AUTOSAR provides a General Specification on Basic Software modules [9, SWS BSW General], which is also valid for CAN Interface.

Thus, the specification SWS BSW General shall be considered as additional and required specification for CAN Interface.

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 [1]. 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 [2, Specification of CAN Transceiver Driver]. 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 (see Figure 5.1).

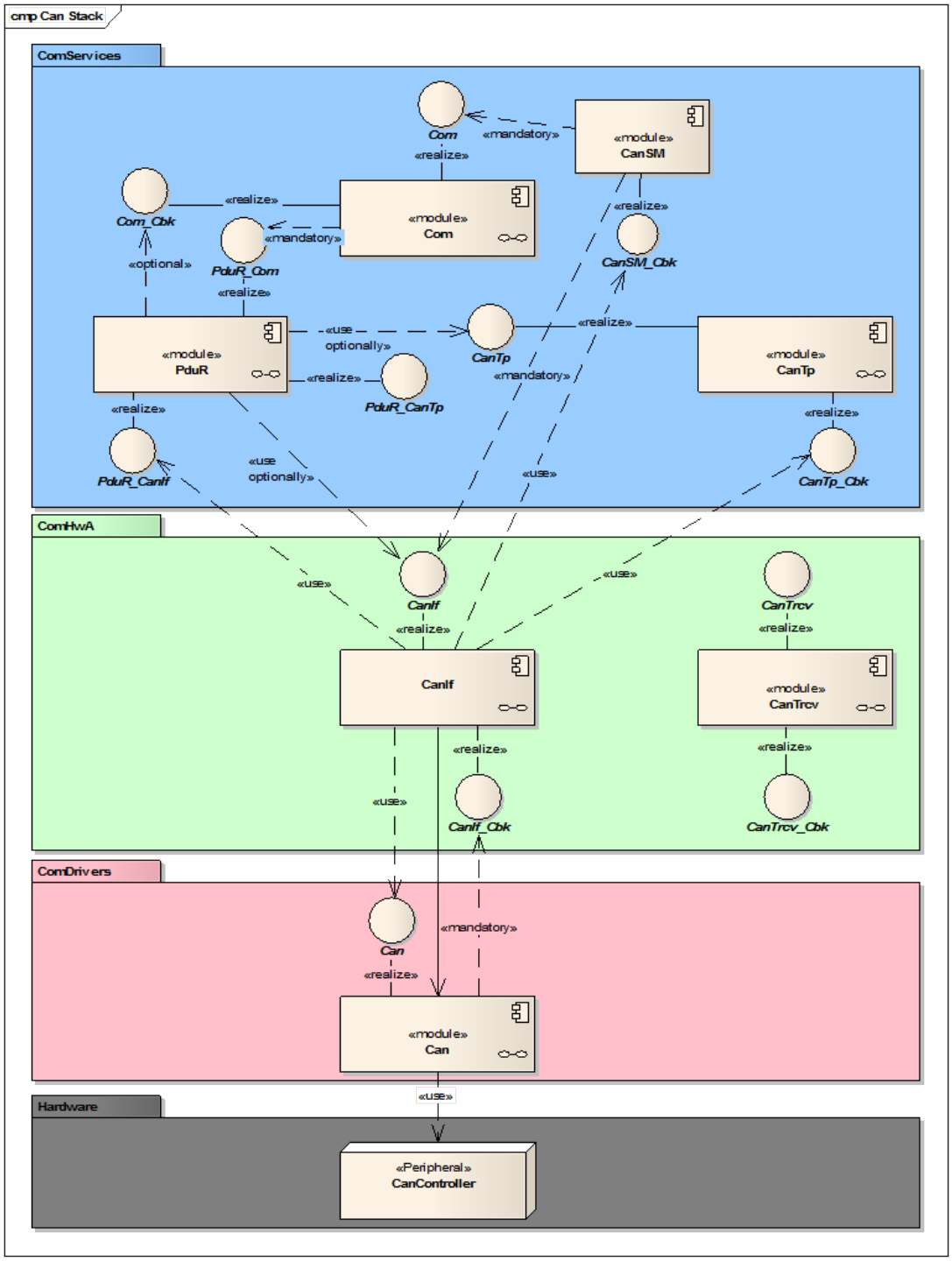

Figure 5.1: CANIF dependencies in AUTOSAR BSW

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), Complex Driver modules (Abbr.: *CDD*), Universal Calibration Protocol module (Abbr.: *XCP*), J1939 Transport Layer module (Abbr.: J1939Tp) and J1939 Network Management module (Abbr.: J1939Nm).

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 section 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 [3, 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 [1, 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.

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

The CanIf provides notification services used by the CanDrv in all notifications scenarios, for example: *transmit confirmation* (subsection 8.4.1 CanIf_TxConfirmation, see [SWS_CANIF_00007]), *receive indication* (subsection 8.4.2 CanIf_RxIndication, see [SWS_CANIF_00006]), *transmit cancellation notification* (subsection 8.4.3 CanIf_CancelTxConfirmation, see [SWS_CANIF_00101]), *BusOff notification* (subsection 8.4.4 CanIf_ControllerBusOff, see [SWS_CANIF_00218]) and *notification of a controller mode change* (subsection 8.4.8, see [SWS_CANIF_00699]).

In case of using multiple CanDrv serving different interrupt vectors these callback services mentioned above must be re-entrant, refer to section 7.25 Multiple CAN Driver support. Reentrancy of callback functions is specified in section 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 section 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 section 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 [2, Specification of CAN Transceiver Driver]).

Each CanTrcv itself does operation mode control of the CAN Transceiver device. The CanIf just maps all APIs of several underlying CanTrcvs 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 [2, 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 [SWS_CANIF_00287]).

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 [SWS_CANIF_00001]) 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 10.1.

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`, *ECUC_CanIf_00625*; `CANIF_HTH_ID_SYMREF`, *ECUC_CanIf_00627*).
- 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 section 7.21 Software receive filter, see `CANIF_SOFTWARE_FILTER_HRH`, `CANIF_HRH_CAN_CONTROLLER_ID_REF`, `CANIF_HRH_ID_SYMREF`, *ECUC_CanIf_00634*)

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

- Transmitting upper layer module and transmit *I-PDU* for each transmit `L-SDU`.
  => Used for dispatching of transmit confirmation services
  (see `CANIF_CANTXPDUID`, *ECUC_CanIf_00247*).
- Receiving upper layer module and receive *I-PDU* for each receive `L-SDU`.
  => Used for `L-SDU` dispatching during receive indication
  (see `CANIF_CANRXPDUID`, *ECUC_CanIf_00249*).

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, *ECUC_CanIf_00248*; CanIfRxPduConfig, *ECUC_CanIf_00249*):

- All L-PDUs received on each physical channel of this ECU.
  => Used for software filtering and receive L-SDU dispatch
- All L-SDUs 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-SDU (i.e. PduR, CanNm, CanTp).
  => Used for the transmit confirmation services
- Receiver for each receive L-SDU (i.e. PduR, CanNm, CanTp)
  => Used for the L-PDU dispatch
- Symbolic L-PDU/L-SDU name.
  => Used for the representation of Rx/Tx data buffer addresses

5.7 File structure

5.7.1 Code file structure

[SWS_CANIF_00377] ⌈ 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. ⌋

[SWS_CANIF_00378] ⌈ 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 [SWS_CANIF_00377] and [SWS_CANIF_00378]: 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 5.2.

5.7.2 Header file structure

[SWS_CANIF_00672] ⌈ 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.

[SWS_CANIF_00643] ⌈ The generic type definitions of the CanIf which are described in section 8.2 shall be performed in the header file CanIf_Types.h. This file has to be included in the header file CanIf.h. ⌋

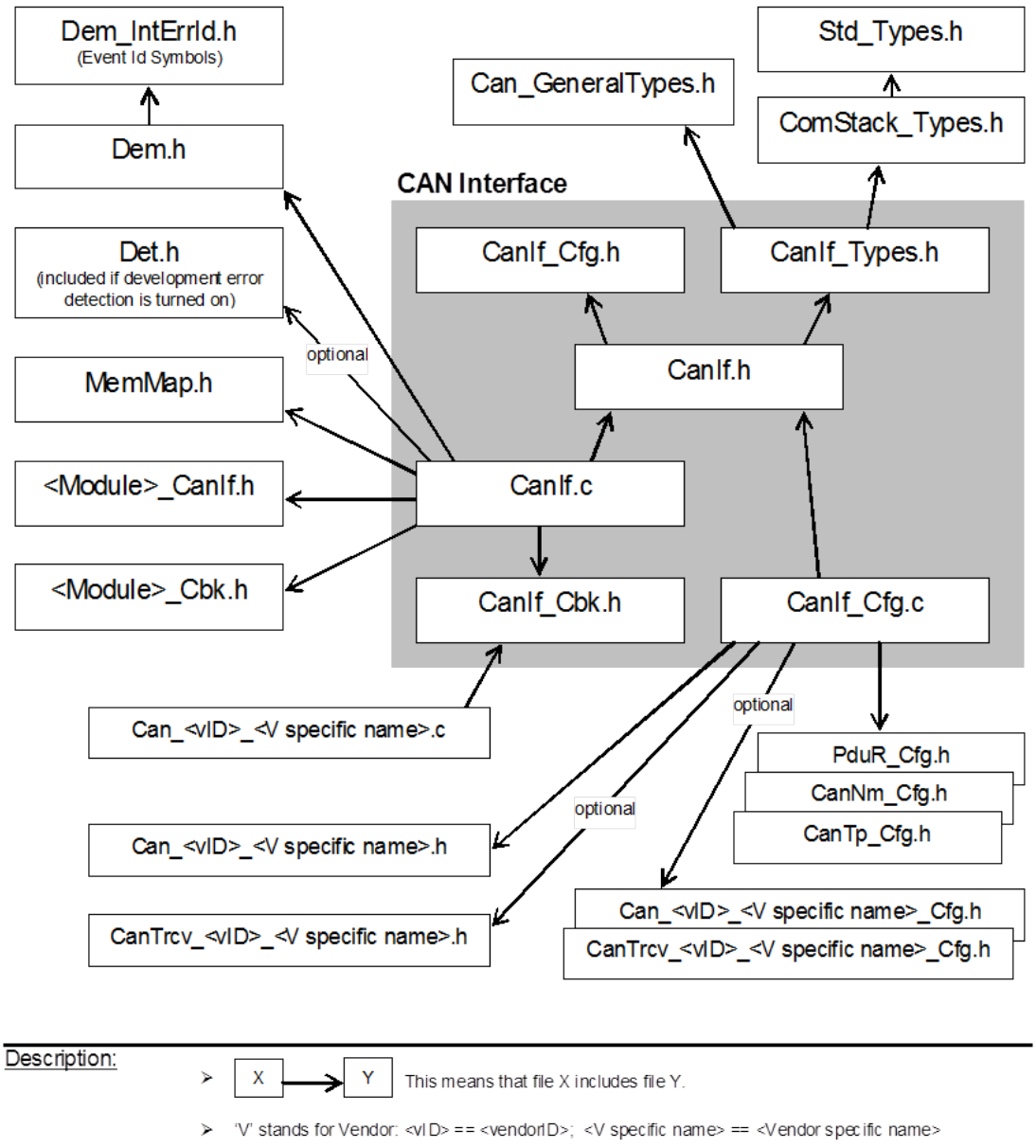

Figure 5.2: Code and include file structure

[SWS_CANIF_00463] ⌈ 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 general CAN stack type declarations
ComStack_Types.h	for COM related type definitions
MemMap.h	for accessing the module specific functionality provided by the BSW Memory Mapping

⌋(*SRS_BSW_00436*)

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

[SWS_CANIF_00208] ⌈ 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

⌋(*SRS_BSW_00415*)

[SWS_CANIF_00233] ⌈ 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 *ECUC_CanIf_00671*)
Xcp_Cbk.h	for callback declarations of the *XCP*
J1939Tp_Cbk.h	for callback declarations of the J1939Tp
J1939Nm_Cbk.h	for callback declarations of the J1939Nm

⌋

[SWS_CANIF_00280] ⌈ 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)

]

`J1939Tp_Cfg.h`	for J1939Tp configuration data (e.g. J1939Tp target PDU Ids)
`J1939Nm_Cfg.h`	for J1939Nm configuration data (e.g. J1939Nm target PDU Ids)

6 Requirements Tracing

The following tables references the requirements specified in [10] as well as [11] and links to the fulfillment of these. Please note that if column 'Satisfied by' is empty for a specific requirement this means that this requirement is not fulfilled by this document.

Requirement	Description	Satisfied by
[BSW00431]	No description	[SWS_CANIF_00999]
[BSW00434]	No description	[SWS_CANIF_00999]
[BSW01024]	No description	[SWS_CANIF_00999]
[SRS_BSW_00007]	No description	[SWS_CANIF_00999]
[SRS_BSW_00010]	No description	[SWS_CANIF_00999]
[SRS_BSW_00101]	No description	[SWS_CANIF_00001]
[SRS_BSW_00164]	No description	[SWS_CANIF_00999]
[SRS_BSW_00167]	No description	[SWS_CANIF_00999]
[SRS_BSW_00168]	No description	[SWS_CANIF_00999]
[SRS_BSW_00170]	No description	[SWS_CANIF_00999]
[SRS_BSW_00172]	No description	[SWS_CANIF_00999]
[SRS_BSW_00306]	No description	[SWS_CANIF_00999]
[SRS_BSW_00307]	No description	[SWS_CANIF_00999]
[SRS_BSW_00308]	No description	[SWS_CANIF_00999]
[SRS_BSW_00309]	No description	[SWS_CANIF_00999]
[SRS_BSW_00326]	No description	[SWS_CANIF_00999]
[SRS_BSW_00328]	No description	[SWS_CANIF_00999]
[SRS_BSW_00330]	No description	[SWS_CANIF_00999]
[SRS_BSW_00334]	No description	[SWS_CANIF_00999]
[SRS_BSW_00336]	No description	[SWS_CANIF_00999]
[SRS_BSW_00341]	No description	[SWS_CANIF_00999]
[SRS_BSW_00342]	No description	[SWS_CANIF_00462]
[SRS_BSW_00353]	No description	[SWS_CANIF_00142]
[SRS_BSW_00358]	No description	[SWS_CANIF_00001]
[SRS_BSW_00361]	No description	[SWS_CANIF_00142]
[SRS_BSW_00373]	No description	[SWS_CANIF_00999]
[SRS_BSW_00376]	No description	[SWS_CANIF_00999]
[SRS_BSW_00378]	No description	[SWS_CANIF_00999]
[SRS_BSW_00404]	No description	[SWS_CANIF_00462]
[SRS_BSW_00411]	No description	[SWS_CANIF_00158]
[SRS_BSW_00414]	No description	[SWS_CANIF_00001]
[SRS_BSW_00416]	No description	[SWS_CANIF_00999]
[SRS_BSW_00417]	No description	[SWS_CANIF_00999]
[SRS_BSW_00423]	No description	[SWS_CANIF_00999]
[SRS_BSW_00424]	No description	[SWS_CANIF_00999]
[SRS_BSW_00425]	No description	[SWS_CANIF_00999]
[SRS_BSW_00426]	No description	[SWS_CANIF_00999]
[SRS_BSW_00427]	No description	[SWS_CANIF_00999]
[SRS_BSW_00428]	No description	[SWS_CANIF_00999]
[SRS_BSW_00429]	No description	[SWS_CANIF_00999]
[SRS_BSW_00432]	No description	[SWS_CANIF_00999]
[SRS_BSW_00433]	No description	[SWS_CANIF_00999]

Requirement	Description	Satisfied by
[SRS_BSW_00435]	No description	[SWS_CANIF_00999]
[SRS_CAN_01014]	No description	[SWS_CANIF_00999]
[SRS_CAN_01021]	No description	[SWS_CANIF_00001]
[SRS_CAN_01022]	No description	[SWS_CANIF_00001]
[SRS_CAN_01129]	No description	[SWS_CANIF_00194]
[SRS_CAN_01131]	No description	[SWS_CANIF_00230]
[SRS_CAN_01139]	No description	[SWS_CANIF_00999]
[SRS_BSW_00159]	All modules of the AUTOSAR Basic Software shall support a tool based configuration	[SWS_CANIF_00999]
[SRS_BSW_00312]	Shared code shall be reentrant	[SWS_CANIF_00064]
[SRS_BSW_00323]	All AUTOSAR Basic Software Modules shall check passed API parameters for validity	[SWS_CANIF_00302] [SWS_CANIF_00311] [SWS_CANIF_00313] [SWS_CANIF_00319] [SWS_CANIF_00320] [SWS_CANIF_00325] [SWS_CANIF_00326] [SWS_CANIF_00331] [SWS_CANIF_00336] [SWS_CANIF_00341] [SWS_CANIF_00346] [SWS_CANIF_00352] [SWS_CANIF_00353] [SWS_CANIF_00364] [SWS_CANIF_00398] [SWS_CANIF_00404] [SWS_CANIF_00410] [SWS_CANIF_00416] [SWS_CANIF_00417] [SWS_CANIF_00419] [SWS_CANIF_00424] [SWS_CANIF_00429] [SWS_CANIF_00535] [SWS_CANIF_00536] [SWS_CANIF_00537] [SWS_CANIF_00538] [SWS_CANIF_00648] [SWS_CANIF_00649] [SWS_CANIF_00650] [SWS_CANIF_00652] [SWS_CANIF_00656] [SWS_CANIF_00657] [SWS_CANIF_00774] [SWS_CANIF_00828] [SWS_CANIF_00860] [SWS_CANIF_00869] [SWS_CANIF_00872] [SWS_CANIF_00873]
[SRS_BSW_00325]	The runtime of interrupt service routines and functions that are running in interrupt context shall be kept short	[SWS_CANIF_00135]

Requirement	Description	Satisfied by
[SRS_BSW_00344]	BSW Modules shall support link-time configuration	[SWS_CANIF_00460] [SWS_CANIF_00461] [SWS_CANIF_00462]
[SRS_BSW_00348]	All AUTOSAR standard types and constants shall be placed and organized in a standard type header file	[SWS_CANIF_00142]
[SRS_BSW_00405]	BSW Modules shall support multiple configuration sets	[SWS_CANIF_00001]
[SRS_BSW_00407]	Each BSW module shall provide a function to read out the version information of a dedicated module implementation	[SWS_CANIF_00158]
[SRS_BSW_00415]	Interfaces which are provided exclusively for one module shall be separated into a dedicated header file	[SWS_CANIF_00208]
[SRS_BSW_00436]	No description	[SWS_CANIF_00463]
[SRS_CAN_01001]	No description	[SWS_CANIF_00023]
[SRS_CAN_01003]	No description	[SWS_CANIF_00012]
[SRS_CAN_01005]	No description	[SWS_CANIF_00026]
[SRS_CAN_01008]	No description	[SWS_CANIF_00005]
[SRS_CAN_01009]	No description	[SWS_CANIF_00007]
[SRS_CAN_01011]	No description	[SWS_CANIF_00068]
[SRS_CAN_01015]	No description	[SWS_CANIF_00104]
[SRS_CAN_01018]	No description	[SWS_CANIF_00030]
[SRS_CAN_01020]	No description	[SWS_CANIF_00063]
[SRS_CAN_01027]	No description	[SWS_CANIF_00003]
[SRS_CAN_01028]	No description	[SWS_CANIF_00229]
[SRS_CAN_01029]	No description	[SWS_CANIF_00014]
[SRS_CAN_01114]	No description	[SWS_CANIF_00033]
[SRS_CAN_01125]	No description	[SWS_CANIF_00194]
[SRS_CAN_01126]	No description	[SWS_CANIF_00381] [SWS_CANIF_00382]
[SRS_CAN_01130]	No description	[SWS_CANIF_00202] [SWS_CANIF_00230]
[SRS_CAN_01136]	No description	[SWS_CANIF_00179]
[SRS_CAN_01140]	No description	[SWS_CANIF_00281]
[SRS_CAN_01141]	No description	[SWS_CANIF_00243]

Document: General Requirements on Basic Software Modules [10]

[SRS_BSW_00344] Reference to link-time configuration	[SWS_CANIF_00461], [SWS_CANIF_00462]
[SRS_BSW_00404] Reference to post build time configuration	[SWS_CANIF_00462]
[SRS_BSW_00405] Reference to multiple configuration sets	[SWS_CANIF_00001], subsection 8.2.1 CanIf_ConfigType
[SRS_BSW_00345] Pre-Build Configuration	Fulfilled by configuration parameter definitions in chapter 10. The configuration parameters are described in a general way.
[SRS_BSW_00159] Tool-based configuration	Not applicable

	(assigned to configuration tool)
[SRS_BSW_00167] Static configuration checking	Not applicable (assigned to configuration tool)
[SRS_BSW_00171] Configurability of optional functionality	Fulfilled by configuration parameter definitions in chapter 10. The configuration parameters are described in a general way.
[SRS_BSW_00170] Data for reconfiguration of SW-components	Not applicable (no interface to AUTOSAR SW Components)
[SRS_BSW_00380] Separate C-Files for configuration parameters	[*SWS_CANIF_00374*], [*SWS_CANIF_00376*]
[SRS_BSW_00419] Separate C-Files for pre-compile time configuration parameters	[*SWS_CANIF_00376*]
[SRS_BSW_00381] Separate configuration header file for pre-compile time parameters	[*SWS_CANIF_00122*]
[SRS_BSW_00412] Separate H-File for configuration parameters	[*SWS_CANIF_00122*]
[SRS_BSW_00383] List dependencies of configuration files	subsection 5.7.2 Header file structure
[SRS_BSW_00384] List dependencies to other modules	chapter 5 Dependencies to other modules, section 5.4 Lower layers: CAN Driver
[SRS_BSW_00387] Specify the configuration class of call-out function	Fulfilled by API definitions in chapter 8.
[SRS_BSW_00388] Introduce containers	Fulfilled by configuration parameter definitions in chapter 10.
[SRS_BSW_00389] Containers shall have names	Fulfilled by configuration parameter definitions in chapter 10.
[SRS_BSW_00390] Parameter content shall be unique within the module	Fulfilled by configuration parameter definitions in chapter 10.
[SRS_BSW_00391] Parameter shall have unique names	Fulfilled by configuration parameter definitions in chapter 10.
[SRS_BSW_00392] Parameters shall have a type	Fulfilled by configuration parameter definitions in chapter 10.
[SRS_BSW_00393] Parameters shall have a range	Fulfilled by configuration parameter definitions in chapter 10.
[SRS_BSW_00394] Specify the scope of the parameters	Fulfilled by configuration parameter definitions in chapter 10.
[SRS_BSW_00395] List the required parameters (per parameter)	Fulfilled by configuration parameter definitions in chapter 10.
[SRS_BSW_00396] Configuration classes	Fulfilled by configuration parameter definitions in chapter 10.
[SRS_BSW_00397] Pre-compile-time parameters	Fulfilled by configuration parameter definitions in chapter 10.
[SRS_BSW_00398] Link-time parameters	Fulfilled by configuration parameter definitions in chapter 10.
[SRS_BSW_00399] Loadable Post-build time parameters	Fulfilled by configuration parameter definitions in chapter 10.
[SRS_BSW_00400] Selectable Post-build time parameters	Fulfilled by configuration parameter definitions in chapter 10.
[SRS_BSW_00402] Published information	[*SWS_CANIF_00725*]

[SRS_BSW_00375] Notification of wake-up reason	[*CANIF013*]
[SRS_BSW_00101] Initialization interface	[SWS_CANIF_00001]
[SRS_BSW_00416] Sequence of Initialization	Not applicable (no initialization dependencies for this module)
[SRS_BSW_00406] Check module initialization	Fulfilled by API definitions in chapter 8.
[SRS_BSW_00168] Diagnostic Interface of SW components	Not applicable (this module does not support a special diagnostic interface)
[SRS_BSW_00407] Function to read out published parameters	[SWS_CANIF_00158]
[SRS_BSW_00423] Usage of SW-C template to describe BSW modules with AUTOSAR Interfaces	Not applicable (this module does not provide an AUTOSAR interface)
[SRS_BSW_00424] BSW main processing function task allocation	Not applicable (requirement on system design, not on a single module)
[SRS_BSW_00425] Trigger conditions for schedulable objects	Not applicable (requirement on system configuration, not on a single module)
[SRS_BSW_00426] Exclusive areas in BSW modules	Not applicable (no exclusive areas specified for this module)
[SRS_BSW_00427] ISR description for BSW modules	Not applicable (this module does not provide any ISRs)
[SRS_BSW_00428] Execution order dependencies of main processing functions	Not applicable (No scheduled API)
[SRS_BSW_00429] 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)
[SRS_BSW_00432] Modules should have separate main processing functions for read/receive and write/transmit data path	Not applicable (requirement on the CAN Driver module)
[SRS_BSW_00433] 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)
[SRS_BSW_00336] Shutdown interface	Not applicable (architecture decision)
[SRS_BSW_00337] Classification of errors	Table in section 7.27 Error classification
[SRS_BSW_00338] Detection and Reporting of development errors	[*SWS_CANIF_00019*]
[SRS_BSW_00369] Do not return development error codes via API	[*SWS_CANIF_00018*]
[SRS_BSW_00339] Reporting of production relevant error status	[*SWS_CANIF_00020*]
[SRS_BSW_00417] Reporting of Error Events by Non-Basic Software (this is a basic software module)	Not applicable

[SRS_BSW_00323] API parameter checking	[SWS_CANIF_00302], [SWS_CANIF_00311], [SWS_CANIF_00313], [SWS_CANIF_00319], [SWS_CANIF_00320], [SWS_CANIF_00325], [SWS_CANIF_00326], [SWS_CANIF_00331], [SWS_CANIF_00336], [SWS_CANIF_00341], [SWS_CANIF_00346], [SWS_CANIF_00352], [SWS_CANIF_00353], [SWS_CANIF_00364], [SWS_CANIF_00398], [SWS_CANIF_00404], [SWS_CANIF_00410], [SWS_CANIF_00416], [SWS_CANIF_00417], [SWS_CANIF_00168], [SWS_CANIF_00419], [SWS_CANIF_00424], [SWS_CANIF_00429], [SWS_CANIF_00535], [SWS_CANIF_00536], [SWS_CANIF_00537], [SWS_CANIF_00538], [SWS_CANIF_00648], [SWS_CANIF_00649], [SWS_CANIF_00650], [SWS_CANIF_00652], [SWS_CANIF_00656], [SWS_CANIF_00657]
[SRS_BSW_00004] Version check	[*SWS_CANIF_00021*]
[SRS_BSW_00409] Header files for production code error IDs	[*SWS_CANIF_00153*]
[SRS_BSW_00385] List possible error notifications	Table in section 7.27 Error classification
[SRS_BSW_00386] Configuration for detecting an error	[*SWS_CANIF_00018*], [*SWS_CANIF_00019*], [*SWS_CANIF_00156*]
[SRS_BSW_00161] Microcontroller abstraction	section 5.6 Configuration
[SRS_BSW_00162] ECU layout abstraction	section 5.6 Configuration
[SRS_BSW_00005] No hard coded horizontal interfaces within MCAL	section 5.7 File structure
[SRS_BSW_00415] User dependent include files	[SWS_CANIF_00208]
[SRS_BSW_00164] Implementation of interrupt service routines	Not applicable
[SRS_BSW_00325] Runtime of interrupt service routines	[SWS_CANIF_00135] The runtime is not totally under control of the CAN Interface, because they are called to the upper layers.
[SRS_BSW_00326] 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)
[SRS_BSW_00342] Usage of source code and object code	[SWS_CANIF_00462] (post build configuration)
[SRS_BSW_00343] Specification and configuration of time	Not applicable (no internal scheduling policy)
[SRS_BSW_00160] Human-readable configuration data	Fulfilled by configuration parameter definitions in chapter 10. The configuration parameters are described in a general way.
[SRS_BSW_00007] HIS MISRA C	Not applicable (requirement on implementation, not on specification)
[SRS_BSW_00300] Module naming convention	Fulfilled by API definitions in chapter 8.

[SRS_BSW_00413] Accessing instances of BSW modules	Fulfilled by API definitions in chapter 8.
[SRS_BSW_00305] Self-defined data types naming convention	Fulfilled by type definitions in section 8.2.
[SRS_BSW_00307] Global variables naming convention	Not applicable (requirement on implementation, not on specification)
[SRS_BSW_00310] API naming convention	Fulfilled by API definitions in chapter 8.
[SRS_BSW_00373] Main processing function naming convention	Not applicable (No scheduled API)
[SRS_BSW_00327] Error values naming convention	Table in section 7.27 Error classification
[SRS_BSW_00335] Status values naming convention	subsection 8.2.3 CanIf_PduModeType, subsection 8.2.4 CanIf_NotifStatusType
[SRS_BSW_00350] Development error detection keyword	[*SWS_CANIF_00019*]
[SRS_BSW_00408] Configuration parameter naming convention	Fulfilled by configuration parameter definitions in chapter 10.
[SRS_BSW_00410] Compiler switches shall have defined values	Fulfilled by configuration parameter definitions in chapter 10.
[SRS_BSW_00411] Get version info keyword	[SWS_CANIF_00158]
[SRS_BSW_00346] Basic set of module files	section 5.7 File structure
[SRS_BSW_00158] Separation of configuration from implementation	section 5.7 File structure
[SRS_BSW_00314] Separation of interrupt frames and service routines	Not applicable (this module does not provide any ISRs)
[SRS_BSW_00370] Separation of call-out interface from API	section 5.7 File structure
[SRS_BSW_00435] Module Header File Structure for the Basic Software Scheduler	Not applicable
[SRS_BSW_00436] Module Header File Structure for the Basic Software Memory Mapping	[SWS_CANIF_00463]
[SRS_BSW_00348] Standard type header	[SWS_CANIF_00142]
[SRS_BSW_00353] Platform specific type header	[SWS_CANIF_00142] (automatically included with Standard types)
[SRS_BSW_00361] Compiler specific language extension header	[SWS_CANIF_00142] (automatically included with Standard types)
[SRS_BSW_00301] Limit imported information	section 5.7 File structure
[SRS_BSW_00302] Limit exported information	[*SWS_CANIF_00116*]
[SRS_BSW_00328] Avoid duplication of code	Not applicable (requirement on implementation, not on specification)
[SRS_BSW_00312] Shared code shall be reentrant	[SWS_CANIF_00064]
[SRS_BSW_00006] Platform independency	Fulfilled by API definitions in section 8.3
[SRS_BSW_00357] Standard API return type	Fulfilled by API definitions in section 8.3.
[SRS_BSW_00377] Module Specific API return type	subsection 8.2.3 CanIf_PduModeType, subsection 8.2.4 CanIf_NotifStatusType

[SRS_BSW_00304] AUTOSAR integer data types	Fulfilled by type and API definitions in section 8.1 and 8.2
[SRS_BSW_00355] Do not redefine AUTOSAR integer data types	Fulfilled by type and API definitions in section 8.1 and 8.2
[SRS_BSW_00378] AUTOSAR Boolean type	Not applicable (no Boolean types used)
[SRS_BSW_00306] Avoid direct use of compiler and platform specific keywords	Not applicable (requirement on implementation, not on specification)
[SRS_BSW_00308] Definition of global data	Not applicable (requirement on implementation, not on specification)
[SRS_BSW_00309] Global data with read-only constraint	Not applicable (requirement on implementation, not on specification)
[SRS_BSW_00371] Do not pass function pointers via API	Fulfilled by API definitions in section 8.3
[SRS_BSW_00358] Return type of `init()` functions	[SWS_CANIF_00001]
[SRS_BSW_00414] Parameter of init function	[SWS_CANIF_00001]
[SRS_BSW_00376] Return type and parameters of main processing functions	Not applicable
[SRS_BSW_00359] Return type of call-out functions	Fulfilled by call-out APIs in section 8.4.
[SRS_BSW_00360] Parameters of call-out functions	Fulfilled by call-out APIs in section 8.4.
[SRS_BSW_00329] Avoidance of generic interfaces	No generic interface used The content of functions might be configuration dependent. The scope of function is always defined
[SRS_BSW_00330] Usage of macros instead of functions	Not applicable (requirement on implementation, not on specification)
[SRS_BSW_00331] Separation of error and status values	section 7.27 Error classification, subsection 8.2.2 CanIf_ControllerModeType, subsection 8.2.4 CanIf_NotifStatusType
[SRS_BSW_00009] Module User Documentation	Fulfilled by the complete documentation.
[SRS_BSW_00401] Documentation of multiple instances of configuration parameters	Fulfilled by configuration parameter definitions in chapter 10.
[SRS_BSW_00172] Compatibility and documentation of scheduling strategy	Not applicable (no internal scheduling policy)
[SRS_BSW_00010] Memory resource documentation	Not applicable (requirement on implementation, not on specification)
[SRS_BSW_00333] Documentation of callback function context	Fulfilled by callback functions in section 8.4.
[SRS_BSW_00374] Module vendor identification	[*SWS_CANIF_00726*]
[SRS_BSW_00379] Module identification	[*SWS_CANIF_00727*]
[SRS_BSW_00003] Version identification	[*SWS_CANIF_00021*]

[SRS_BSW_00318] Format of module version	[*SWS_CANIF_00728*]
[SRS_BSW_00321] Enumeration of module version numbers	[*SWS_CANIF_00729*]
[SRS_BSW_00341] Microcontroller compatibility documentation	Not applicable (no microcontroller dependent module)
[SRS_BSW_00334] Provision of XML file	Not applicable (requirement on implementation, not on specification)

Document: Requirements on CAN [12]

Requirement	Satisfied by
[SRS_CAN_01033] Basic Software General Requirements	Fulfilled by this chapter.
[SRS_CAN_01125] Data throughput read direction	[SWS_CANIF_00194]
[SRS_CAN_01126] Data throughput write direction	[SWS_CANIF_00381], [SWS_CANIF_00382]
[SRS_CAN_01139] CAN Controller specific Initialization	Not applicable
[SRS_CAN_01129] Receive Data Interface for CAN Interface and CAN Driver Module	section 7.16 Read received data, subsection 8.3.6 CanIf_ReadRxPduData, [SWS_CANIF_00194]
[SRS_CAN_01121] Interfaces of the CAN Interface module	section 5.4 Lower layers: CAN Driver, section 5.5 Lower layers: CAN Transceiver Driver
[SRS_CAN_01014] Network configuration abstraction	Not applicable
[SRS_CAN_01001] HW independence	[SWS_CANIF_00023]
[SRS_CAN_01015] Network Database Information Import	[SWS_CANIF_00104]
[SRS_CAN_01016] Interface to CAN Driver configuration	section 10.2
[SRS_CAN_01018] Software Filter	[SWS_CANIF_00030]
[SRS_CAN_01019] DLC Check configuration	section 10.2
[SRS_CAN_01020] `Tx Buffer` configuration	[SWS_CANIF_00063]
[SRS_CAN_01021] CAN Interface Module Power-On Initialization	[SWS_CANIF_00001]
[SRS_CAN_01022] Dynamic selection of static configuration sets	[SWS_CANIF_00001]
[SRS_CAN_01023] Power-On Initialization Sequence	section 7.8
[SRS_CAN_01002] Rx PDU dispatching	[*CANIF024*]
[SRS_CAN_01003] Reception indication dispatcher	[SWS_CANIF_00012]
[SRS_CAN_01114] Data Consistency of transmit L-PDUs	[SWS_CANIF_00033]
[SRS_CAN_01004] Software Filtering for L-PDU reception	section 7.21
[SRS_CAN_01005] DLC check for L-PDU reception	[SWS_CANIF_00026]

Requirement	Satisfied by
[SRS_CAN_01006] Rx L-PDU enable/disable	[CANIF096]
[SRS_CAN_01007] Tx L-PDU dispatching	[*CANIF024*]
[SRS_CAN_01008] Transmission request service	[SWS_CANIF_00005]
[SRS_CAN_01009] Transmission confirmation service	[SWS_CANIF_00007]
[SRS_CAN_01011] Tx buffering	[SWS_CANIF_00068]
[SRS_CAN_01013] Tx L-PDU enable/disable service	[CANIF096]
[SRS_CAN_01027] CAN Controller Mode Select service	[SWS_CANIF_00003]
[SRS_CAN_01028] CAN Controller State Service	[SWS_CANIF_00229]
[SRS_CAN_01032] Wake-up Notification	[*CANIF013*]
[SRS_CAN_01061] Dynamic Tx Handles	section 7.4
[BSW01024] DLC Error Notification	Not applicable
[SRS_CAN_01029] Bus-off notification	[SWS_CANIF_00014]
[SRS_CAN_01130] Read Status Interface of CAN Interface	[SWS_CANIF_00202], [SWS_CANIF_00230]
[SRS_CAN_01131] Mixed mode of notification and polling mechanism	[SWS_CANIF_00230]
[SRS_CAN_01136] Notification of first received CAN message	[SWS_CANIF_00179]
[SRS_CAN_01129] Receive Data Interface for CAN Interface	[SWS_CANIF_00194]
[SRS_CAN_01140] Support of Standard and Extended Identifiers	[SWS_CANIF_00281]
[SRS_CAN_01141] Support of both Standard and Extended Identifiers on one network (optional feature)	[SWS_CANIF_00243], [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 an event occurs. In this case the relevant CanIf services are 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 a *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.

[SWS_CANIF_00023] ⌈ The CanIf shall avoid direct access to hardware specific communication buffers and shall access it exclusively via the CanDrv interface services. ⌋*(SRS_CAN_01001)*

Rationale for [SWS_CANIF_00023]: 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.

[SWS_CANIF_00662] ⌈ The CanIf shall use two types of HOHs to enable access to the CanDrv:

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

⌋

[SWS_CANIF_00291] ⌈ Definition of HRH: The HRH shall be a handle referencing a logical Hardware Receive Object of the CAN Controller mailbox. ⌋

[SWS_CANIF_00665] ⌈ The HRH shall enable CanIf to use *BasicCAN* or a *FullCAN* reception method of the referenced reception unit and to indicate a Received L-SDU to a target upper layer module. ⌋

[SWS_CANIF_00663] ⌈ If the HRH references a reception unit configured for *BasicCAN transmission*, software filtering shall be enabled in the CanIf. ⌋

[SWS_CANIF_00664] ⌈ If multiple HRHs are used, each HRH shall belong at least to a single or fixed group of Rx L-SDU 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.

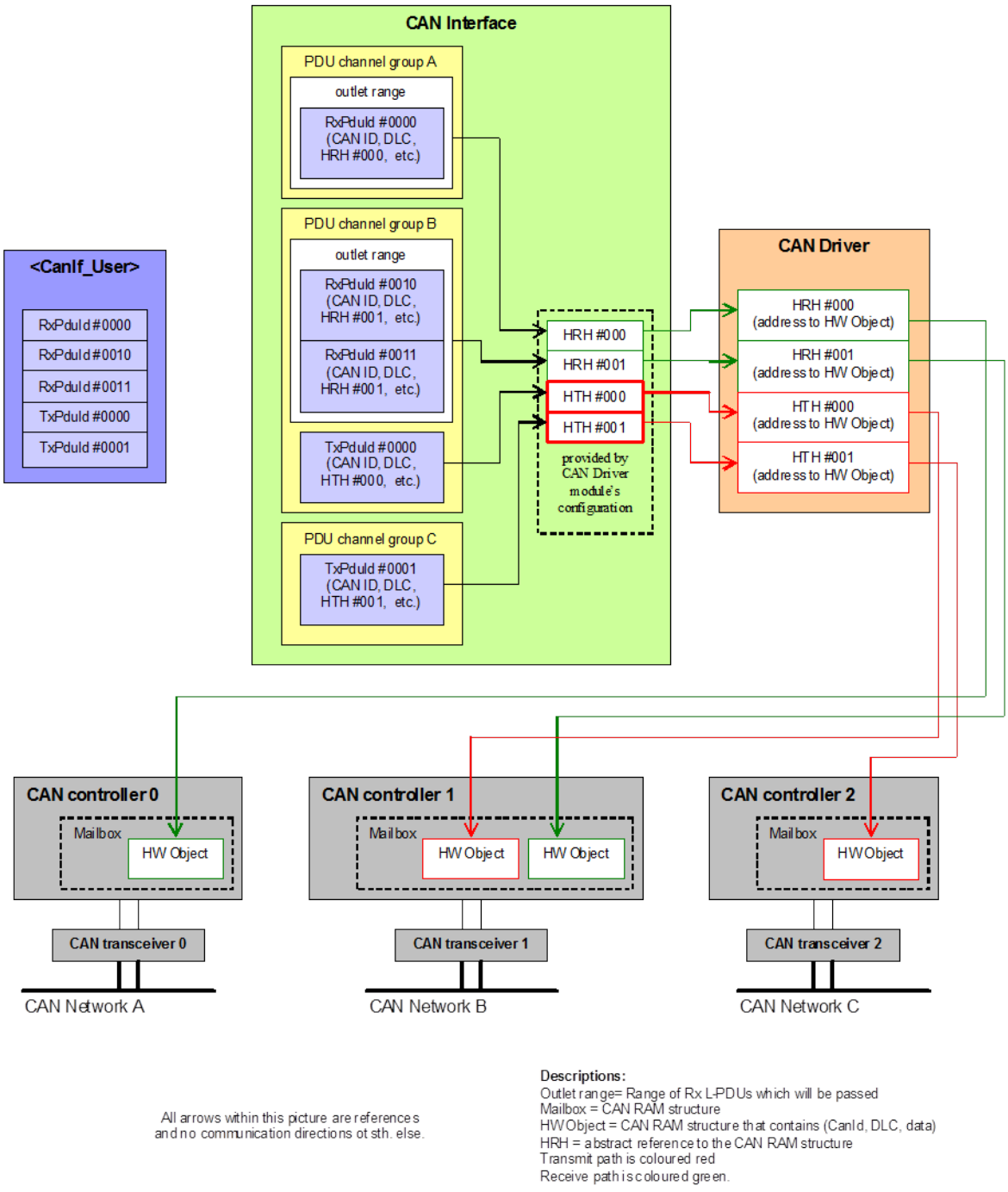

Figure 7.1: Mapping between PDU Ids and HW object handles

[SWS_CANIF_00292] ⌈ Definition of HTH: The HTH shall be a handle referencing a logical Hardware Transmit Object of the CAN Controller mailbox. ⌋

[SWS_CANIF_00666] ⌈ The HTH shall enable CanIf to use *BasicCAN* or *FullCAN* transmission method of the referenced transmission unit and to confirm a transmitted L-SDU to a target upper layer module. ⌋

[SWS_CANIF_00466] ⌈ Each CanIf Tx L-PDU shall statically be assigned to one CanIfTxBuffer (see *ECUC_CanIf_00832*) configuration container at configuration time (see *ECUC_CanIf_00831*). ⌋

Rationale for [SWS_CANIF_00466]: CanIf Tx L-PDUs do not refer HTHs, but CanIfTxBuffer, which in turn do refer HTHs.

[SWS_CANIF_00667] ⌈ If multiple HTHs are used, each HTH shall belong to a single or fixed group of Tx L-PDU handles (CanTxPduIds). ⌋

[SWS_CANIF_00115] ⌈ The CanIf shall be able to use all HRHs and HTHs of one CanDrv as common, single numbering area starting with zero. ⌋

The dedicated HRHs 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

CanIf offers general access to the CAN L-SDU related data for upper layers. The L-SDU Handle facilitates this access. The L-PDU Handle refers to data structures, which consists of CanIf and *CAN PCI* specific attributes describing the L-PDU/L-SDU. 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 *ECUC_CanIf_00619*)	CAN Identifier (CanId) CANIF_TXPDU_CANID (see *ECUC_CanIf_00592*), range of CanIds per *PDU* (see *ECUC_CanIf_00743*), CANIF_RXPDU_CANID (see *ECUC_CanIf_00598*), CANIF_RXPDU_CANID_MASK (see *ECUC_CanIf_00822*)
Direction of *L-PDU* (Tx, Rx) CANIF_TXPDU_ID (see *ECUC_CanIf_00591*), CANIF_RXPDU_ID (see *ECUC_CanIf_00597*)	Type of CAN Identifier (*StandardCAN*, *ExtendedCAN*) referenced from CanDrv via CANIF_HTH_ID_SYMREF (see *ECUC_CanIf_00627*), CANIF_HRH_ID_SYMREF (see *ECUC_CanIf_00634*)

CAN Hardware Unit CANIF_PUBLIC_NUMBER_OF_CAN_HW_UNITS (see *ECUC_CanIf_00615*)	*Data Length Code* (*DLC*) CANIF_TXPDU_DLC (see *ECUC_CanIf_00594*), CANIF_RXPDU_DLC (see *ECUC_CanIf_00599*)
HTH/HRH of the CAN Controller	Reference to the PDU data (see [1, Specification of CAN Driver])
Target ID for the corresponding upper layer CANIF_TXPDU_USERTXCONFIRMATION (see *ECUC_CanIf_00527*), CANIF_RXPDU_USERRXINDICATION_UL (see *ECUC_CanIf_00529*)	
Type of Transmit L-PDU Handle (*static*, *dynamic*) CANIF_TXPDU_TYPE (see *ECUC_CanIf_00593*)	
Type of Tx/Rx L-PDU (*FullCAN*, *BasicCAN*) CANIF_HTH_ID_SYMREF (see *ECUC_CanIf_00627*), CANIF_HRH_ID_SYMREF (see *ECUC_CanIf_00634*)	

[SWS_CANIF_00046] ⌈ 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 [SWS_CANIF_00046]: This relation is used in order to ensure correct *L-SDU* dispatching at transmission confirmation and reception indication events. In this manner CanIf is able to identify the CAN Controller 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 7.20.2 PDU channel modes, see CanIf_SetPduMode(), [SWS_CANIF_00008]). 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 [7, 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 L-SDU ID (CanTxPduId/CanRxPduId) defined by the CanIf as parameter.
- For all callback APIs, which are invoked by 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 L-PDU/L-SDU Id of the callee.

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

7.4 Dynamic CAN L-PDU handles

CanIf shall support the ability to filter incoming messages using the CanIdMask. The filtering shall be done by comparing the incoming CanId with the stored CanId after applying the CanIdMask to both IDs. This should be done after the filtering of regular CanIds without mask, to allow for separate handling of some of the CanIds that fall into the range defined by the mask or a CanId based range.

Additionally, "dynamic" Tx and Rx L-SDUs shall be supported, where parts of the CanId reside in the MetaData of the L-SDU.

During transmission of dynamic L-SDUs, when a CanIdMask is defined, the variable parts of the CanId provided via the MetaData must be merged with the CanId by using this mask. When no CanIdMaskand no CanId are configured, the MetaData shall be used directly as CanId. In this case, the MetaDataLength of the L-SDU must be large enough to contain the whole CanId.

During reception of dynamic L-SDUs, the lower <MetaDataLength> bytes of the received CanId shall be placed in the L-SDU MetaData (in *little endian byte order*), while the L-SDU length is incremented accordingly. The layout of the MetaData is independent of the CanIdMask parameter. For efficiency reasons, the ID could already be placed at the end of the data by CanDrv.

[SWS_CANIF_00844] ⌈ CanIf shall support dynamic L-PDU Handles, where the CanId or parts of the CanId are placed in the MetaData of a L-SDU, which resides in the data buffer directly behind the payload data. The number of ID bytes in the payload data is defined by the parameter MetaDataLength of the global PDUs referenced by CanIfTxPduRef or CanIfRxPduRef. The L-SDU length is set to the sum of the payload length and MetaDataLength. ⌋

[SWS_CANIF_00845] ⌈ The sequence of the CanId bytes in the MetaData is *little endian*, i.e. the lowest byte of the ID (the 8 least significant bits) is placed in the first byte after the actual L-SDU data. ⌋

[SWS_CANIF_00846] ⌈ If MetaDataLength is smaller than the actual CanId size, the highest bytes of the CanId shall be omitted. If MetaDataLength is larger than the CanId size, the space after the ID bytes shall be padded with zeros. ⌋

7.4.1 Dynamic transmit L-PDU handles

Definition of dynamic Transmit L-PDUs: L-PDUs which allow reconfiguration of the CanId during runtime (CANIF_TXPDU_TYPE == DYNAMIC) or where the ID or parts thereof are provided as MetaData of the L-SDU (MetaDataLength => 1).

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 *ECUC_CanIf_00527*) 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() (see [SWS_CANIF_00189]) reconfigures the CanId of a dynamic L-PDU with CANIF_TXPDU_TYPE == DYNAMIC (see CANIF593_Conf).

[SWS_CANIF_00188] ⌈ CanIf shall process the two most significant bits of the CanId (see [1, Specification of CAN Driver], definition of Can_IdType [*SWS_Can_00416*]) to determine which type of CanId is used and thus how the dynamic Transmit L-PDU shall be transmitted. ⌋

[SWS_CANIF_00673] ⌈ 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 [SWS_CANIF_00064]. ⌋

[SWS_CANIF_00853] ⌈ If MetaDataLength is smaller than the actual CanId size, the parameters CanIfTxPduCanIdMask and CanIfTxPduCanId must be configured. ⌋

[SWS_CANIF_00855] ⌈ If MetaDataLength is at least as large as the actual CanId size, CanIfTxPduCanIdMask and CanIfTxPduCanId can be omitted. In this case, the CanId is directly taken from the MetaData. ⌋

[SWS_CANIF_00856] ⌈ CanIfTxPduCanIdMask shall be ignored when MetaDataLength is not configured for this L-SDU. ⌋

[SWS_CANIF_00854] ⌈ If MetaDataLength, CanIfTxPduCanIdMask and CanIfTxPduCanId are available, CanIfTxPduCanIdMask defines the bits in CanIfTxPduCanId that shall appear in the actual CanId, the other bits are taken from the MetaData. ⌋

Note: The resulting ID could be calculated in the following way: (CanIfTxPduCanId & CanIfTxPduCanIdMask) | (<dynamic ID parts> & ~CanIfTxPduCanIdMask)

[SWS_CANIF_00857] ⌈ CanIf_Init() (see [SWS_CANIF_00085]) initializes the CanIds of the dynamic Transmit L-PDUs with CANIF_TXPDU_TYPE == DYNAMIC (see CANIF593_Conf) to the value configured via CanIfTxPduCanId. ⌋

7.4.2 Dynamic receive L-PDU handles

Definition of dynamic Receive L-PDUs: L-PDUs that correspond to a set of CanIds, where the actually received CanId is provided to upper layers as part of the PDU data.

[SWS_CANIF_00847] ⌈ Configuration shall ensure that dynamic Receive L-PDUs use an ID range or a mask and that the MetaData is configured for the L-SDU. Besides this, the software filtering must be enabled for these L-SDUs. ⌋

[SWS_CANIF_00848] ⌈ Upon reception of a dynamic L-SDU, CanIf shall ensure that <MetaDataLength> bytes of the CanId are placed in the MetaData, and shall increase the L-SDU length accordingly. ⌋

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 channel. Those APIs are used by the CanSm to provide a network view to the ComM (see [3]) 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 [SWS_CANIF_00014]).

[SWS_CANIF_00653] ⌈ 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 *ECUC_CanIf_00647*). ⌋

Example:

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

[SWS_CANIF_00655] ⌈ 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 *ECUC_CanIf_00654*). ⌋

Example:

CanIf	CanDrv A	CanDrv 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`.

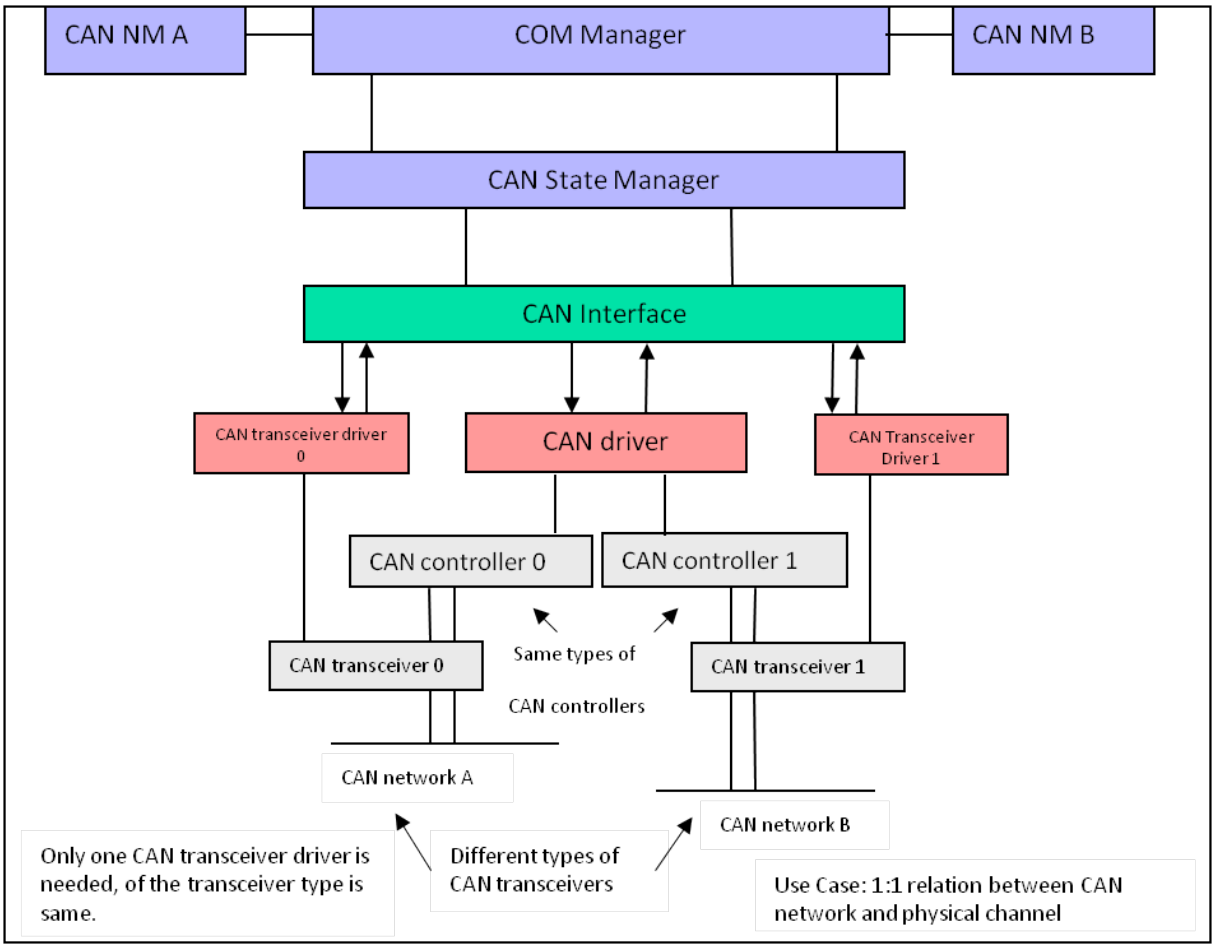

Figure 7.2: Physical channel view definition example A

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.

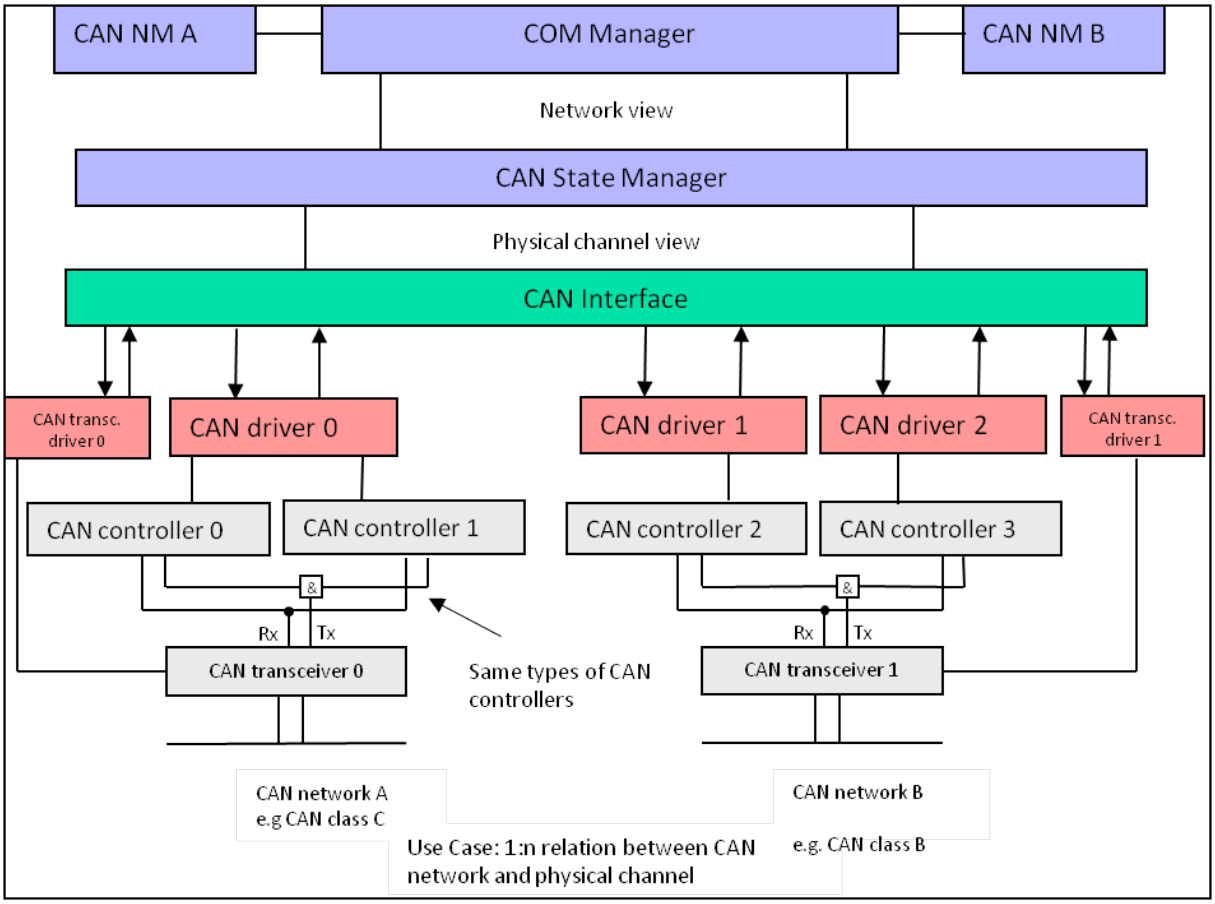

Figure 7.3: Physical channel view definition example B

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 section 7.2 Hardware object handles and section 7.25 Multiple CAN Driver support).

Figure 7.4 shows a CAN Hardware Unit consisting of two CAN Controllers of the same type connected to two physical channels:

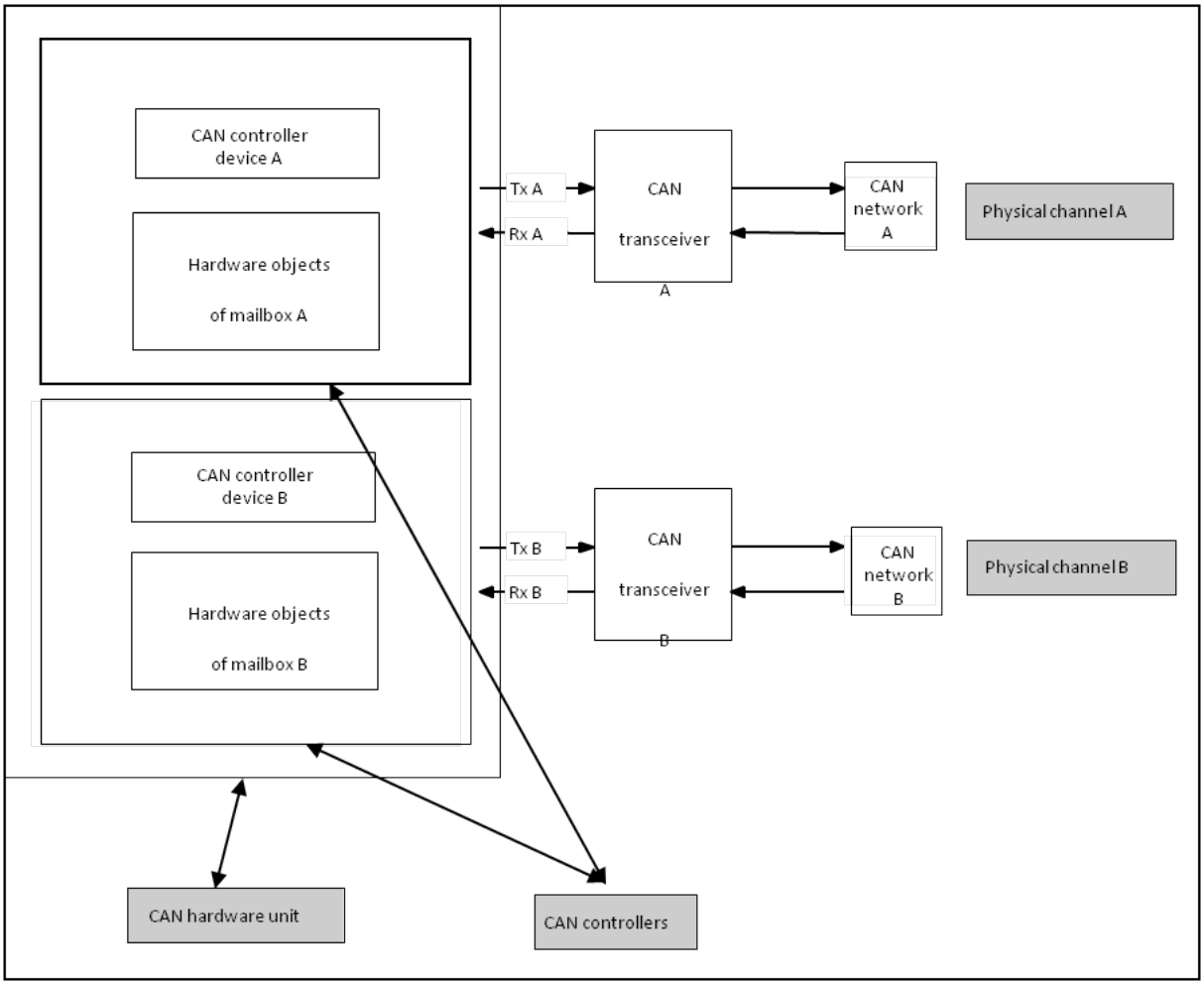

Figure 7.4: Typical CAN Hardware Unit

7.7 BasicCAN and FullCAN reception

`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, `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 section 7.21 Software receive filter.

[SWS_CANIF_00467] ⌈ CanIf shall configure and store an order on HTHs and HRHs for all HOHs derived from the configuration containers CanIfHthCfg (see *ECUC_CanIf_00258*) and CanIfHrhCfg (see *ECUC_CanIf_00259*) ⌋

[SWS_CANIF_00468] ⌈ CanIf shall reference a hardware acceptance filter for each HOH derived from the configuration parameters CANIF_HTH_ID_SYMREF (see *ECUC_CanIf_00627*) and CANIF_HRH_ID_SYMREF (see *ECUC_CanIf_00634*). ⌋

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

[SWS_CANIF_00469] ⌈ 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 *ECUC_CanIf_00632*). ⌋

[SWS_CANIF_00211] ⌈ CanIf shall execute the software acceptance filter from [SWS_CANIF_00469] 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.

[SWS_CANIF_00877] ⌈ CanIf shall use the *MSB* of Mailbox->CanId (Can_IdType) passed by CanDrv in order to select the correct RxPdu specified by CanIfRxPduCfg. While CanIfRxPduCanId has to match the CAN ID, CanIfRxPduCanIdType has to match the *MSB* of the received Can_IdType. ⌋

Basically, CanIf supports reception either of *Standard CAN IDs* or *Extended CAN IDs* on one Physical CAN Channel by the parameters CANIF_TXPDU_CANIDTYPE (see *ECUC_CanIf_00590*) and CANIF_RXPDU_CANIDTYPE (see *ECUC_CanIf_00596*).

[SWS_CANIF_00281] ⌈ CanIf shall accept and handle *StandardCAN IDs* and *ExtendedCAN IDs* on the same Physical Channel (= mixed mode operation). ⌋*(SRS_CAN_01140)*

In a mixed mode operation *Standard CAN IDs* and *Extended CAN 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_TXPDU_CANIDTYPE (see *ECUC_CanIf_00590*) and CANIF_RXPDU_CANIDTYPE (see *ECUC_CanIf_00596*). In case of mixed mode operation the software acceptance filter algorithm (see section 7.21 Software receive filter) must be able to deal with both type of CanIds.

[SWS_CANIF_00281] 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 [SWS_CANIF_00001]). 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 [1] and [2, Specification of CAN Transceiver Driver]).

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 [SWS_CANIF_00001]).

If re-initialization of the entire CAN modules during runtime is required, the EcuM shall invoke the CanSm (see [3]) 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 subsection 8.6.3).

7.9 Transmit request

CanIf's transmit request function CanIf_Transmit() ([SWS_CANIF_00005]) is a common interface for upper layers to transmit L-PDUs on the CAN network. The upper communication layer modules initiate the transmission only via CanIf's services without direct access to CanDrv. The initiated Transmit Request is successfully completed, if 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 subsection 8.3.4 CanIf_Transmit).

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

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

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

[SWS_CANIF_00382] ⌈ If an L-PDU is requested to be transmitted via a PDU channel mode (refer to subsection 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. ⌋(*SRS_CAN_01126*)

[SWS_CANIF_00723] ⌈ If an L-PDU is requested to be transmitted via a CAN Controller, whose CCMSM (see section 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 section 7.12 Transmit confirmation 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-SDU specific data is organized by the following parameters:

- Transmit L-PDU Handle => L-SDU ID
- Reference to a data structure, which contains L-SDU related data: L-SDU length (1) and pointer to the L-SDU (2), including MetaData for dynamic Transmit L-PDUs handle when MetaDataLength is configured for that L-SDU.

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

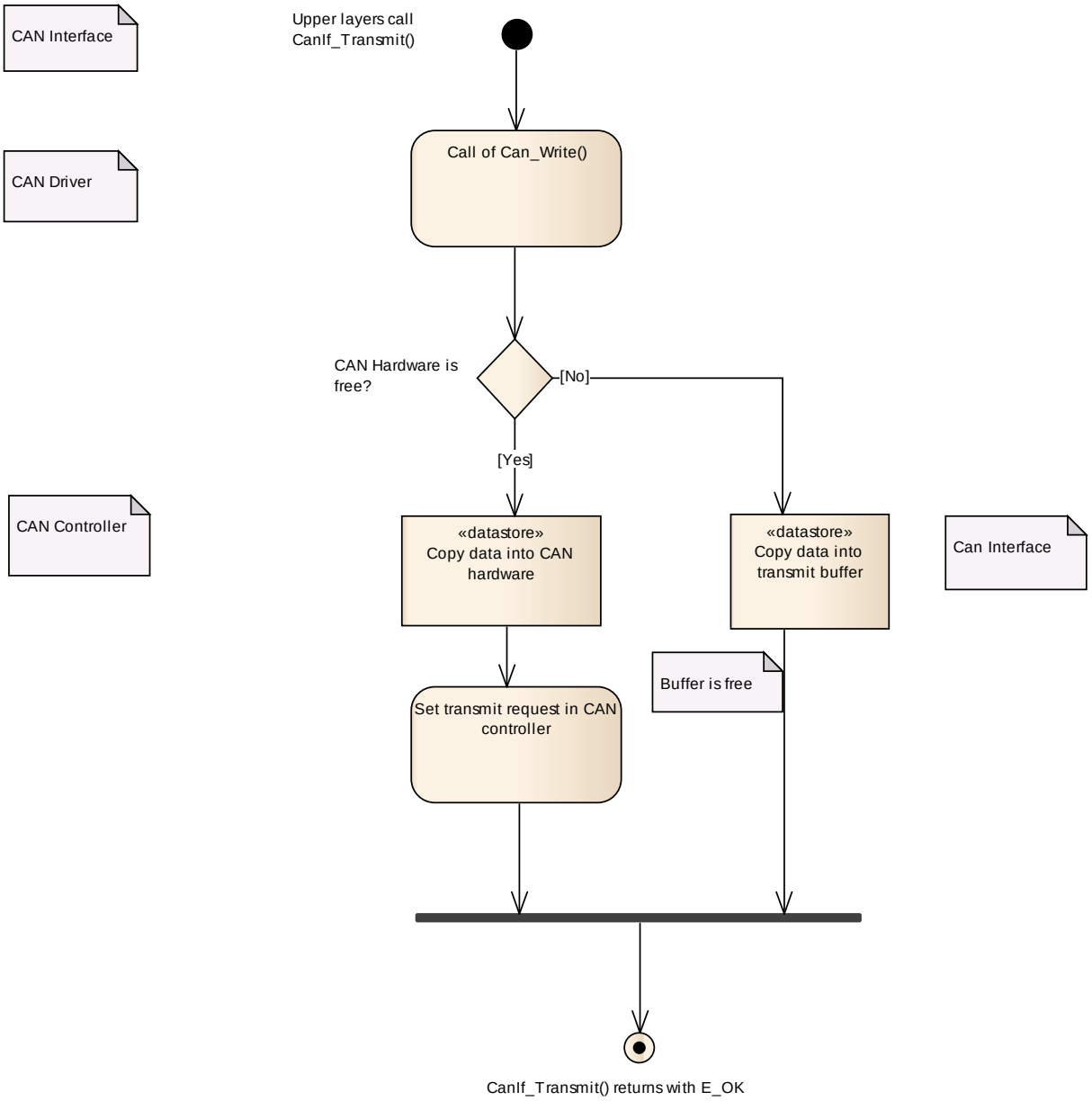

Figure 7.5: Transmit data flow

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 [SWS_CANIF_00318]).

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 Can-

Drv 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 *ECUC_CanIf_00832*). Whereas each CanIfTxBuffer is assigned to one or multiple dedicated HTH's (see *ECUC_CanIf_00833*) 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 [SWS_CANIF_00835] a CanIf Tx L-PDU refers HTHs via the CanIfTxBuffer configuration container (see *ECUC_CanIf_00832*). This is valid if transmit buffering is not needed as well. In this case, the buffer size (see *ECUC_CanIf_00834*) 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

ECUC_CanIf_00831, *ECUC_CanIf_00832*, *ECUC_CanIf_00833* and *ECUC_CanIf_00834* 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 [SWS_CANIF_00381]) or
- a pending transmit request was successfully aborted (see [SWS_CANIF_00054]).

[SWS_CANIF_00063] ⌈ The CanIf shall support buffering of a CAN L-PDU Handle for *BasicCAN* transmission in the CanIf, if parameter CANIF_PUBLIC_TX_BUFFERING (see *ECUC_CanIf_00618*) is enabled. ⌋*(SRS_CAN_01020)*

[SWS_CANIF_00849] ⌈ For dynamic Transmit L-PDU Handles, also the CanId has to be stored in the CanIfTxBuffer. ⌋

[SWS_CANIF_00381] ⌈ If transmit buffering is enabled (see [SWS_CANIF_00063]) 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. ⌋*(SRS_CAN_01126)*

When the call of Can_Write() returns with CAN_BUSY, the CanDrv has rejected the requested transmission of the L-PDU (see [1]) because there is no free HW object available at time of the transmit request (Tx request).

[SWS_CANIF_00835] ⌈ When the CanIf checks whether it is possible to buffer a CanIf Tx L-PDU (see [SWS_CANIF_00381], [SWS_CANIF_00054]), this shall only be possible, if the CanIf Tx L-PDU is assigned (see *ECUC_CanIf_00831*) to a CanIfTxBuffer (see *ECUC_CanIf_00832*), which is configured with a buffer size (see *ECUC_CanIf_00834*) 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 *ECUC_CanIf_00834*).

[SWS_CANIF_00836] ⌈ If it is possible to buffer a CanIf Tx L-PDU, because the buffer size of the assigned CanIfTxBuffer is bigger than zero (see [SWS_CANIF_00835]), 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. ⌋

[SWS_CANIF_00068] ⌈ If it is possible to buffer a CanIf Tx L-PDU, because the buffer size of the assigned CanIfTxBuffer is bigger than zero (see [SWS_CANIF_00835]), 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. ⌋*(SRS_CAN_01011)*

[SWS_CANIF_00068] implies that a CanIf Tx L-PDU shall not be overwritten in a CanIfTxBuffer in the context of `CanIf_CancelTxConfirmation()`

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 [1] for further details.

[SWS_CANIF_00837] ⌈ If the buffer size is greater zero, all buffer elements are busy and `CanIf_Transmit()` is called with a new `L-PDU` (no other instance of the same Pdu is already stored in the buffer), then the new `L-PDU` shall not be stored and `CanIf_Transmit()` shall return `E_NOT_OK`. ⌋

7.11.2.2 Clearance of transmit L-PDU buffers

[SWS_CANIF_00386] ⌈ The CanIf shall evaluate during transmit confirmation (see [SWS_CANIF_00007]), whether pending CanIf Tx L-PDUs are stored within the CanIfTxBuffers, which are assigned to the new free Hardware Transmit Object (see [SWS_CANIF_00466]). ⌋

[SWS_CANIF_00668] ⌈ If pending CanIf Tx L-PDUs are available in the CanIfTxBuffers as per [SWS_CANIF_00386], 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 [SWS_CANIF_00070]) by call of `Can_Write()`. ⌋

[SWS_CANIF_00070] ⌈ The CAN Interface module shall transmit L-PDUs stored in the transmit L-PDU buffers in priority order (see [13]) per each HTH. ⌋

[SWS_CANIF_00183] ⌈ 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 [SWS_CANIF_00183] 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

[SWS_CANIF_00387] ⌈ When function `CanIf_Init()` is called, CanIf shall initialize every transmit L-PDU buffer assigned to the CanIf. ⌋

The requirement [SWS_CANIF_00387] is necessary to prevent transmission of old data after restart of the CAN Controller.

7.11.3 Data integrity of transmit L-PDU buffers

[SWS_CANIF_00033] ⌈ CanIf shall protect against concurrent access to transmit L-PDU buffers for transmit L-PDUs. ⌋*(SRS_CAN_01114)*

This may be realized by using 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().

Rationale: for [SWS_CANIF_00033]: 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, CanDrv notifies it to CanIf by the call of CanIf_TxConfirmation() ([SWS_CANIF_00007]).

[SWS_CANIF_00383] ⌈ When callback notification CanIf_TxConfirmation() is called, CanIf shall identify the upper layer communication layer (see [SWS_CANIF_00414]), 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 section 7.12 Transmit 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 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 subsubsection 8.6.3.1 <User_TxConfirmation>.

[SWS_CANIF_00740] ⌈ If CANIF_PUBLIC_TXCONFIRM_POLLING_SUPPORT (see *ECUC_CanIf_00733*) is enabled, CanIf shall buffer the information about a

received TxConfirmation per CAN Controller, if the CCMSM of that controller is in state CANIF_CS_STARTED. ⌋

7.12.1.1 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. CanDrv informs CanIf about a successful transmit cancellation via CanIf_CancelTxConfirmation() (see subsection 8.4.3 CanIf_CancelTxConfirmation).

[SWS_CANIF_00054] ⌈ When CanIf_CancelTxConfirmation() is called, CanIf shall check if it is possible to buffer the canceled CanIf Tx L-PDU, which is referenced in parameter CanPduPtr of CanIf_CancelTxConfirmation(), inside a CanIfTxBuffer. ⌋

For further information about the CanIfTxBuffer see section 7.11 Transmit buffering.

7.13 Transmit cancellation

CanIf shall execute transmissions of all pending transmit requests in the transmit L-PDU buffers in priority order (see [SWS_CANIF_00070]). 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 [1]). 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

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

- Priority Inversion of the L-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

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

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

[SWS_CANIF_00176] ⌈ 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 subsubsection 7.11.2.1 Storage 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 CanIf's transmit L-PDU buffers.

Hint: 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.

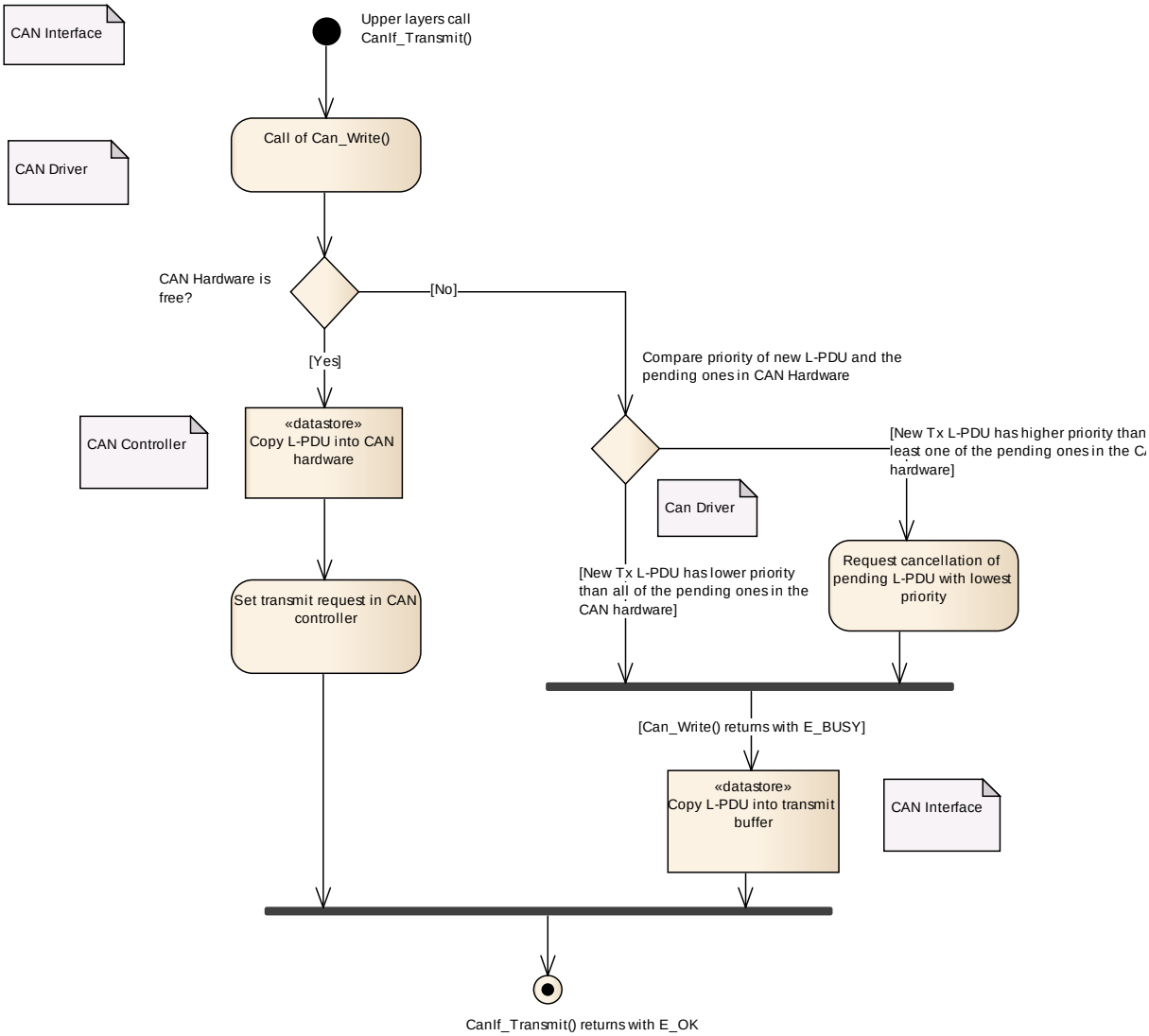

Figure 7.6: Transmit cancellation request

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 [1] for more details.

Transmit cancellation can be enabled and disabled by configuration (configuration parameter `CANIF_TX_CANCELLATION`, see *ECUC_CanIf_00640*). This feature can be activated only, as far as transmit `L-PDU` buffers have been enabled (configuration parameter `CANIF_PUBLIC_TX_BUFFERING`, see *ECUC_CanIf_00618*). 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 *ECUC_CanIf_00640*).

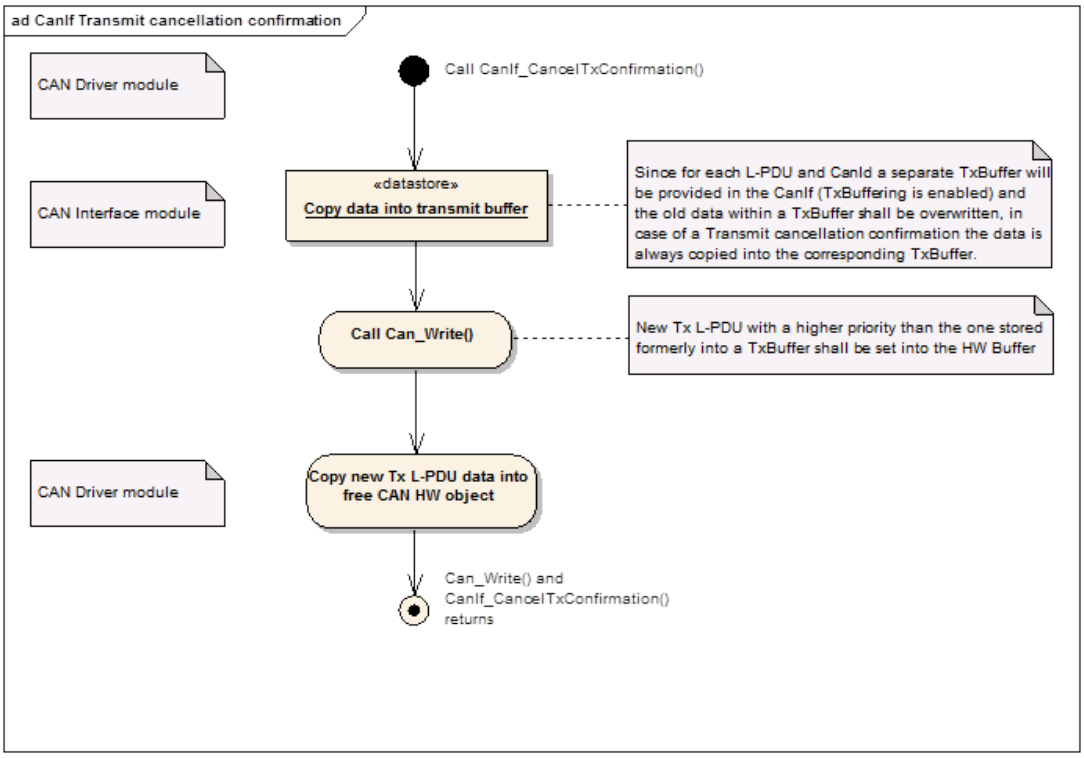

Figure 7.7: Transmit cancellation confirmation

7.14 Receive data flow

According to the AUTOSAR Basic Software Architecture the received data will be evaluated and processed in the upper layer communication stacks (i.e. AUTOSAR COM, CanNm, CanTp, DCM). This means, upper layer modules may neither work with (i.e. change) buffers of CanDrv (Rx) nor do they have access to buffers of CanIf (Tx).

CanIf provides internal buffering in the receive path only if CANIF_PUBLIC_READRXPDU_DATA_API (see *ECUC_CanIf_00607*) is set to TRUE (refer to section 7.16). Tx buffering is addressed in section 7.11 and dynamic L-PDUs are concerned in section 7.4.

In case of a new reception of an L-PDU CanDrv calls CanIf_RxIndication() (refer to [SWS_CANIF_00006]) of 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 Received L-PDU

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

After the indication of a received L-PDU by CanDrv (CanIf_RxIndication() is called) the CanIf shall proceed as described in 7.15 Receive indication. CanIf is not able to recognize, whether CanDrv uses temporary buffering or a direct hardware access. It 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 CanIf returns to avoid loss of data.

CanDrv, 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 CanDrv.

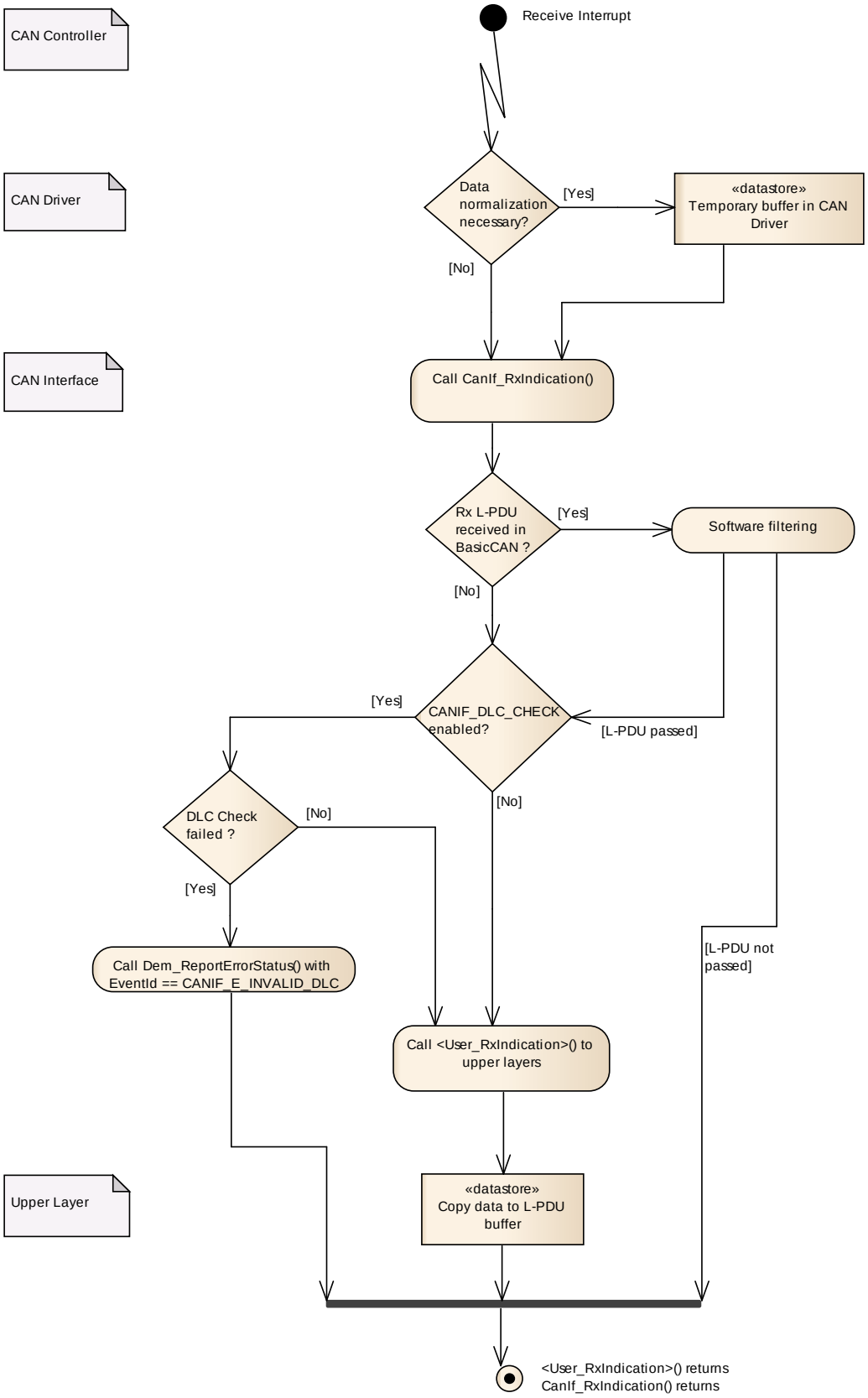

Figure 7.8: Receive data flow

7.15 Receive indication

A call of CanIf_RxIndication() (see [SWS_CANIF_00006]) 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 L-SDU for later access by the upper communication layers. The CanIf notifies upper layer modules about this asynchronous event using <User_RxIndication>() (see subsubsection 8.6.3.2 <User_RxIndication>, [SWS_CANIF_00012]), if configured and if this CAN L-PDU is successfully detected and accepted for further processing. The detailed requirements for this behavior follow here.

[SWS_CANIF_00389] ⌈ 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 *ECUC_CanIf_00628* equals 0..*) If Software Filtering rejects the received L-PDU, the CanIf shall end the receive indication for that call of CanIf_RxIndication(). ⌋

[SWS_CANIF_00390] ⌈ If the CanIf accepts an L-PDU received via CanIf_RxIndication() during Software Filtering (see [SWS_CANIF_00389]), the CanIf shall process the DLC check afterwards, if configured (see *ECUC_CanIf_00617*). ⌋

For further details, please refer to section 7.22 DLC Check.

[SWS_CANIF_00297] ⌈ If the CanIf has accepted a L-PDU received via CanIf_RxIndication() during DLC check (see [SWS_CANIF_00390]), the CanIf shall copy the number of bytes according to the configured DLC value (see *ECUC_CanIf_00594*, *ECUC_CanIf_00599*) to the static receive buffer, if configured for that L-PDU (see [SWS_CANIF_00198], *ECUC_CanIf_00600*). ⌋

[SWS_CANIF_00851] ⌈ If MetaData is configured for a received L-SDU, CanIf shall copy the PDU payload and the CAN ID to the static receive buffer. ⌋

[SWS_CANIF_00056] ⌈ If CanIf accepts a L-PDU received via CanIf_RxIndication() during DLC check (see [SWS_CANIF_00390], [SWS_CANIF_00026]), CanIf shall identify if a target upper layer module was configured (see configuration descrption of [SWS_CANIF_00012] and *ECUC_CanIf_00529*, *ECUC_CanIf_00530*) to be called with its providing receive indication service for the received L-SDU. ⌋

[SWS_CANIF_00135] ⌈ If a target upper layer module was configured to be called with its providing receive indication service (see [SWS_CANIF_00056]), the CanIf shall call this configured receive indication callback service (see *ECUC_CanIf_00530*) and shall provide the parameters required for upper layer notification callback functions (see [SWS_CANIF_00012]) based on the parameters of CanIf_RxIndication(). ⌋*(SRS_BSW_00325)*

Note: A single receive L-PDU can only be assigned to a single receive indication callback service (refer to multiplicity of CANIF_USERRXINDICATION_NAME, *ECUC_CanIf_00530*).

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-SDU if configured
- call upper layer receive indication callback service, if configured.

7.16 Read received data

The read received data API CanIf_ReadRxPduData() (see [SWS_CANIF_00194]) is a common interface for upper layer modules to read CAN L-SDUs recently received from the CAN network. The upper layer modules initiate the receive request only via CanIf services without direct access to CanDrv. The initiated receive request is successfully completed, if CanIf wrote the received L-SDU into the upper layer module I-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, *ECUC_CanIf_00607*), not necessarily a receive indication service for the same L-SDU has to be configured (see *ECUC_CanIf_00529*). If needed, the receive indication can be enabled, too.

By this way the type of mechanism to receive L-SDUs (in the upper layer modules of CanIf) can be chosen at configuration time by the parameter CANIF_RXPDU_USERRXINDICATION_UL (see *ECUC_CanIf_00529*) and parameter CANIF_RXPDU_READ_DATA (see *ECUC_CanIf_00600*) according to the needs of the upper layer module, to which the corresponding receive L-SDU belongs to. For details please refer to section 9.10 Read received data.

[SWS_CANIF_00198] ⌈ If the configuration parameter CANIF_PUBLIC_READRXPDU_DATA_API (*ECUC_CanIf_00607*) is set to TRUE, CanIf shall store each received L-SDU, at which CANIF_RXPDU_READDATA (*ECUC_CanIf_00600*) is enabled, into a receive L-SDU buffer. This means that if the configuration parameter CANIF_RXPDU_READDATA (*ECUC_CanIf_00600*) is set to TRUE, CanIf has to allocate a receive L-SDU buffer for this receive L-SDU. ⌋

[SWS_CANIF_00199] ⌈ After call of CanIf_RxIndication() and passing of software filtering and DLC check, CanIf shall store the received L-SDU in this receive L-SDU buffer. During the call of CanIf_ReadRxPduData() the assigned receive L-SDU buffer containing a recently received L-SDU, CanIf shall avoid preemptive receive L-SDU buffer access events (refer to [SWS_CANIF_00064]) to that receive L-SDU buffer. ⌋

7.17 Read Tx/Rx notification status

In addition to the notification callback functions CanIf provides the API service CanIf_ReadTxNotifStatus() (see [SWS_CANIF_00202]) to read the transmit confirmation status of any transmit L-SDU and the API service CanIf_ReadRxNotifStatus() is provided to read the receive indication status of any receive L-SDU.

CanIf's API services CanIf_ReadTxNotifStatus() (see [SWS_CANIF_00202]) and CanIf_ReadRxNotifStatus() (see [SWS_CANIF_00230]) can be enabled/disabled globally or per L-SDU at pre-compile time configuration using the configuration parameters CANIF_PUBLIC_READTXPDU_NOTIFY_STATUS_API (*ECUC_CanIf_00609*), CANIF_PUBLIC_READRXPDU_NOTIFY_STATUS_API (*ECUC_CanIf_00608*), CANIF_TXPDU_READ_NOTIFYSTATUS (*ECUC_CanIf_00589*), and CANIF_RXPDU_READ_NOTIFYSTATUS (*ECUC_CanIf_00595*).

[SWS_CANIF_00472] ⌈ If configuration parameter CANIF_PUBLIC_READTXPDU_NOTIFY_STATUS_API (*ECUC_CanIf_00609*) is set to TRUE, CanIf shall store the current notification status for each transmit L-SDU. ⌋

[SWS_CANIF_00473] ⌈ If configuration parameter CANIF_PUBLIC_READRXPDU_NOTIFY_STATUS_API (*ECUC_CanIf_00608*) is set to TRUE, CanIf shall store the current notification status for each receive L-SDU. ⌋

Rationale for [SWS_CANIF_00391] and [SWS_CANIF_00393] respectively [SWS_CANIF_00392] and [SWS_CANIF_00394]: 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

[SWS_CANIF_00064] Shared code shall be reentrant ⌈ CanIf shall protect preemptive events, which access shared resources, that could be changed during CanIf's event handling, against each other. ⌋*(SRS_BSW_00312)*

Rationale: An attempt to update the data in the upper layer module buffers as well as in CanIf's internal 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 L-PDUs on the network.

[SWS_CANIF_00058] ⌈ If CanIf's environment reads data from CanIf controlled memory areas initiated by calling one of the functions CanIf_Transmit(),

CanIf_TxConfirmation(), CanIf_CancelTxConfirmation(), and CanIf_ReadRxPduData(), 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 CanIf controlled memory areas only, if CanIf is configured to use transmit buffers or receive buffers.

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

CanIf can e.g. use 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/L-SDU Handles. CanDrv's transmit request API Can_Write() operates re-entrant as well.

7.19 CAN Controller Mode

7.19.1 General Functionality

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 CanIf is 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 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 CanSm. By this way CanSm is responsible to set all CAN Controllers of one CAN network sequentially to sleep mode or to wake them up.

Hint: Because of *CDDs*, the names of the callback services of the Communication Services are configurable (see subsection 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 Figure 7.9, CCMSM) in CanIf to CANIF_CS_STOPPED and which in turn notifies CanSm by the callback service CanSm_ControllerBusOff(ControllerId).

The state machine (CCMSM) in Figure 7.9 gives an overview about the possible CAN Controller State Transitions, which may be requested by surrounding modules of CanIf (CanDrv, CanSm, EcuM, *CDD*, etc.). CanIf does not check these requests for correctness.

CanIf analyses the function calls CanIf_ControllerBusOff() and CanIf_ControllerModeIndication() and determines the current mode of the assigned CAN Controller, which are represented in 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 7.9.

Figure 7.9 shows only one sub-state-machine representing the required behavior of one CAN Controller for sake of lucidity, but there should be a separate sub-state-machine for each assigned CAN Controller.

The calling modules requesting state transitions of the CCMSM can do this independently of the current state of the CCMSM, i.e. CanIf accepts every state transition request by calling the function CanIf_SetControllerMode() or CanIf_ControllerBusOff(). CanIf does not decide if a requested mode transition of the CAN Controller is valid or not. CanIf only includes the execution of requested mode transitions (see [SWS_CANIF_00474]).

This network related state machine is implemented in CanSm. Refer to [3]. CanIf only stores the requested mode and executes the requested transition.

Hint: It has to be regarded that not only CanSm is able to request CAN Controller Mode changes.

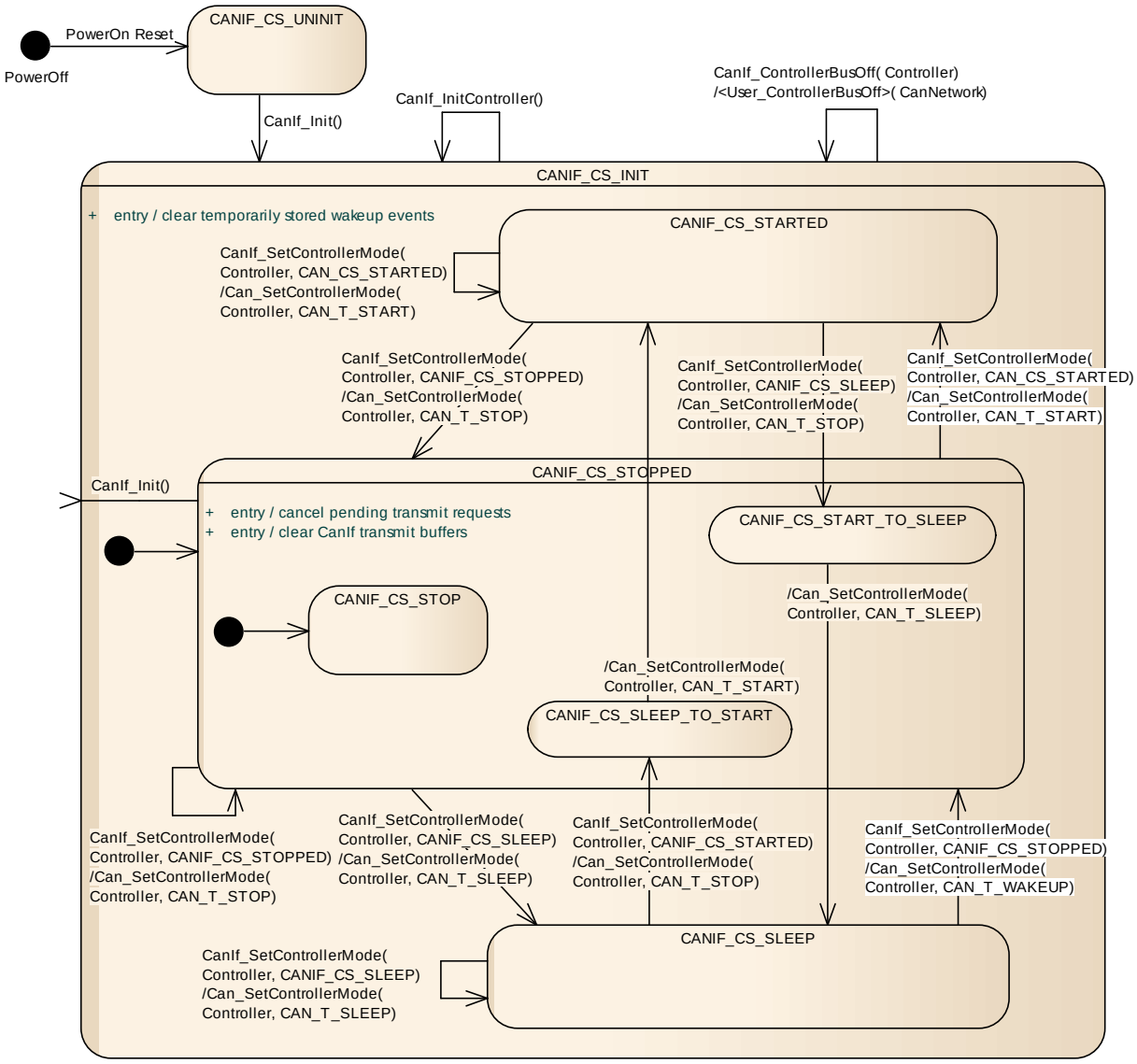

Figure 7.9: CanIf Controller mode state machine for one CAN Controller

General remarks to be considered during implementation:

[SWS_CANIF_00474] ⌈ CanIf shall not contain any complete *CAN Controller State Machine*. ⌋

Hint for [SWS_CANIF_00474]: 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 CanIf's interfaces, the additional states of CCMSM are not mandatory for the implementation of CanIf.

7.19.2 CAN Controller Operation Modes

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

[SWS_CANIF_00475] ⌈ 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

CanIf is not initialized. EcuM has to consider, that also CAN Drivers and CAN Controllers are not initialized.

[SWS_CANIF_00476] ⌈ If a CCMSM is in state CANIF_CS_UNINIT when the function CanIf_Init() is called, then CanIf shall take the CCMSM for every assigned CAN Controller to state CANIF_CS_INIT. ⌋

7.19.2.2 CANIF_CS_INIT

[SWS_CANIF_00477] ⌈ If the CCMSM is in state CANIF_CS_INIT for every assigned CAN Controller when the function CanIf_Init() is called, then CanIf 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 [SWS_CANIF_00477] models the reinitialization of the state machine contained within CANIF_CS_INIT.

[SWS_CANIF_00478] ⌈ If the state CANIF_CS_INIT of a CCMSM is entered, then CanIf shall take that CCMSM to sub state CANIF_CS_STOPPED of state CANIF_CS_INIT. ⌋

[SWS_CANIF_00479] ⌈ If a CCMSM enters state CANIF_CS_INIT, then CanIf shall clear all temporarily stored wakeup events corresponding to that state machine. ⌋

[SWS_CANIF_00298] ⌈ If a CCMSM is in state CANIF_CS_INIT when 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.

[SWS_CANIF_00480] ⌈ If a CCMSM is in state CANIF_CS_STOPPED when CanIf_SetControllerMode(ControllerId, CANIF_CS_STOPPED) is called with parameter ControllerId referencing that CCMSM, then CanIf shall call Can_SetControllerMode(Controller, CAN_T_STOP). ⌋

[SWS_CANIF_00713] ⌈ If a CCMSM is in state CANIF_CS_STOPPED when CanIf_ControllerModeIndication(ControllerId, CANIF_CS_STOPPED) is called with parameter ControllerID referencing that CCMSM, then CanIf shall take the CCMSM to sub state CANIF_CS_STOPPED of state CANIF_CS_INIT. ⌋

[SWS_CANIF_00677] ⌈ 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 [SWS_CANIF_00317]) and returns E_NOT_OK (see [SWS_CANIF_00005]). ⌋

[SWS_CANIF_00481] ⌈ If a CCMSM is in state CANIF_CS_STOPPED when CanIf_SetControllerMode(ControllerId, CANIF_CS_STARTED) is called with parameter ControllerId referencing that CCMSM, then CanIf shall call Can_SetControllerMode(Controller, CAN_T_START). ⌋

[SWS_CANIF_00714] ⌈ If a CCMSM is in state CANIF_CS_STOPPED when CanIf_ControllerModeIndication(ControllerId, CANIF_CS_STARTED) is called with parameter ControllerId referencing that CCMSM, then CanIf shall take the CCMSM to sub state CANIF_CS_STARTED of state CANIF_CS_INIT. ⌋

[SWS_CANIF_00482] ⌈ If a CCMSM is in state CANIF_CS_STOPPED when CanIf_SetControllerMode(ControllerId, CANIF_CS_SLEEP) is called with parameter ControllerId referencing that CCMSM, then CanIf shall call Can_SetControllerMode(Controller, CAN_T_SLEEP). ⌋

[SWS_CANIF_00715] ⌈ If a CCMSM is in state CANIF_CS_STOPPED when CanIf_ControllerModeIndication(ControllerId, CANIF_CS_SLEEP) is called with parameter ControllerId referencing that CCMSM, then CanIf shall take the CCMSM to sub state CANIF_CS_SLEEP of state CANIF_CS_INIT. ⌋

[SWS_CANIF_00485] ⌈ If a CCMSM enters state CANIF_CS_STOPPED, then 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 CanIf passes all transmit requests to corresponding CanDrv and CanIf can receive CAN L-PDUs and notify upper layers about received *L-PDUs*.

[SWS_CANIF_00584] ⌈ If a CCMSM is in state CANIF_CS_STARTED when CanIf_SetControllerMode(ControllerId, CANIF_CS_STARTED) is called with parameter ControllerId referencing that CCMSM, then CanIf shall call Can_SetControllerMode(Controller, CAN_T_START). ⌋

[SWS_CANIF_00716] ⌈ If a CCMSM is in state CANIF_CS_STARTED when CanIf_ControllerModeIndication(ControllerId, CANIF_CS_STARTED) is called with parameter ControllerId referencing that CCMSM, then CanIf shall leave the CCMSM in sub state CANIF_CS_STARTED of state CANIF_CS_INIT. ⌋

[SWS_CANIF_00585] ⌈ If a CCMSM is in state CANIF_CS_STARTED when CanIf_SetControllerMode(ControllerId, CANIF_CS_STOPPED) is called with parameter ControllerId referencing that CCMSM, then CanIf shall call Can_SetControllerMode(Controller, CAN_T_STOP). ⌋

[SWS_CANIF_00717] ⌈ If a CCMSM is in state CANIF_CS_STARTED when CanIf_ControllerModeIndication(ControllerId, CANIF_CS_STOPPED) is called with parameter ControllerId referencing that CCMSM, then the CanIf shall take the CCMSM to sub state CANIF_CS_STOPPED of state CANIF_CS_INIT. ⌋

[SWS_CANIF_00488] ⌈ If a CCMSM is in state CANIF_CS_STARTED when CanIf_ControllerBusOff(ControllerId) is called with parameter ControllerId referencing that CCMSM, then the CCMSM shall be changed to CANIF_CS_STOPPED. ⌋

Note: When a CCMSM is in state CANIF_CS_STARTED, CanIf_SetControllerMode(ControllerId, CANIF_CS_SLEEP) causes an invalid state transition which is detected by CanDrv.

7.19.2.2.3 CANIF_CS_SLEEP

If a CAN Controller does not support a sleep mode, CanDrv will handle corresponding requests with a logical sleep mode (see [1, SWS_Can_00290 in SWS CanDrv]). CanIf is not able to differ between logical and real sleep mode of a CAN Controller.

[SWS_CANIF_00486] ⌈ If a CCMSM is in state CANIF_CS_SLEEP when CanIf_SetControllerMode(ControllerId, CANIF_CS_SLEEP) is called with parameter ControllerId referencing that CCMSM, then CanIf shall call Can_SetControllerMode(Controller, CAN_T_SLEEP). ⌋

[SWS_CANIF_00718] ⌈ If a CCMSM is in state CANIF_CS_SLEEP when CanIf_ControllerModeIndication(ControllerId, CANIF_CS_SLEEP)

is called with parameter ControllerId referencing that CCMSM, then CanIf shall leave the CCMSM in sub state CANIF_CS_SLEEP of state CANIF_CS_INIT. ⌋

[SWS_CANIF_00487] ⌈ If a CCMSM is in state CANIF_CS_SLEEP when CanIf_SetControllerMode(ControllerId, CANIF_CS_STOPPED) is called with parameter ControllerId referencing that CCMSM, then CanIf shall call Can_SetControllerMode(Controller, CAN_T_WAKEUP). ⌋

[SWS_CANIF_00719] ⌈ If a CCMSM is in state CANIF_CS_SLEEP when CanIf_ControllerModeIndication(ControllerId, CANIF_CS_STOPPED) is called with parameter ControllerId referencing that CCMSM, then CanIf shall take the CCMSM to sub state CANIF_CS_STOPPED of state CANIF_CS_INIT. ⌋

Note: When a CCMSM is in state CANIF_CS_SLEEP, CanIf_SetControllerMode(ControllerId, CANIF_CS_STARTED) causes an invalid state transition which is detected by CanDrv.

7.19.2.3 BUSOFF

[SWS_CANIF_00739] ⌈ If CANIF_PUBLIC_TXCONFIRM_POLLING_SUPPORT (see *ECUC_CanIf_00733*) is enabled, CanIf shall clear the information about a TxConfirmation (see [SWS_CANIF_00740]) when callback CanIf_ControllerBusOff(ControllerId) is called. ⌋

[SWS_CANIF_00724] ⌈ When callback CanIf_ControllerBusOff(ControllerId) is called, the CanIf shall call CanSM_ControllerBusOff(ControllerId) of the CanSm (see subsubsection 8.6.3.8 or a *CDD* (see [SWS_CANIF_00559], [SWS_CANIF_00560]). ⌋

Influence on CCMSM of CanIf_ControllerBusOff is described in [SWS_CANIF_00298] and [SWS_CANIF_00488].

7.19.2.4 Mode Indication

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

[SWS_CANIF_00711] ⌈ When callback CanIf_ControllerModeIndication(ControllerId, ControllerMode) is called, CanIf shall call CanSm_ControllerModeIndication(ControllerId, ControllerMode) of the CanSm (see subsubsection 8.6.3.8 <User_ControllerModeIndication>) or a *CDD* (see [SWS_CANIF_00691], [SWS_CANIF_00692]). ⌋

[SWS_CANIF_00712] ⌈ When callback CanIf_TrcvModeIndication(Transceiver, TransceiverMode) is called, CanIf shall call

CanSM_TransceiverModeIndication(TransceiverId, TransceiverMode) of the CanSm (see subsubsection 8.6.3.8 <User_ControllerModeIndication>) or a *CDD* (see [SWS_CANIF_00697], [SWS_CANIF_00698]). ⌋

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 CanDrv calls function CanIf_ControllerModeIndication() and CanIf in turn calls function <User_ControllerModeIndication>() besides changing the CCMSM to CANIF_CS_SLEEP. If CAN 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 Controller Mode within the CanIf buffered operation mode (CCMSM) by CanIf_GetControllerMode() (see [SWS_CANIF_00229]).

Not all types of CAN Controllers support *Sleep* and *Wake-Up Mode*. These modes are then encapsulated by CanDrv by providing hardware independent operation modes via its interface, which has to be managed by CanIf.

Note: It is possible that during transition from CANIF_CS_STOPPED to CANIF_CS_SLEEP CAN Controller may indicate a wake-up interrupt to the ECU Integration Code.

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 CanIf'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 CanDrv or CanTrcv to the ECU Integration Code. For details see respective UML diagram in the chapter "CAN Wakeup Sequences" of document [14].

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 [SWS_CANIF_00180]) CanIf is notified by the Integration Code about a detected CAN wake-up by the service CanIf_CheckWakeup() (see CAN Wakeup Sequences of [14]).

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). CanIf in turn checks by configured input reference to EcuMWakeupSource in CanDrvs, which CanDrvs have to be checked. CanIf gets this information via reference CanIfCtrlCanCtrlRef (see *ECUC_CanIf_00636*).

The Communication Service, which is called, belongs to the service defined during configuration (see *ECUC_CanIf_00250*). In this way EcuM as well as CanSm are able to change CAN Controller States and to control the system behavior concerning the *BusOff recovery* or *wake-up procedure*.

[SWS_CANIF_00180] ⌈ 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 *ECUC_CanIf_00637*) and by CanDrv configuration.
- underlying CAN Transceiver provides *wake-up support* and wake-up is enabled by the parameter CANIF_TRANSCEIVER_WAKEUP_SUPPORT (see *ECUC_CanIf_00606*) and by CanTrcv configuration.

⌋

[SWS_CANIF_00395] ⌈ When CanIf_CheckWakeup(EcuM_WakeupSourceType WakeupSource) is invoked, CanIf shall query CanDrvs / CanTrcvs 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. CanIf just has to find out the exact CAN hardware device.

[SWS_CANIF_00720] ⌈ If at least one function call of Can_CheckWakeup() or CanTrcv_CheckWakeup() returns (CAN_OK / E_OK) to CanIf, then CanIf_CheckWakeup() shall return E_OK. ⌋

[SWS_CANIF_00678] ⌈ If all calls of `Can_CheckWakeup()` or `CanTrcv_CheckWakeup()` return (`CAN_NOT_OK` / `E_NOT_OK`) to `CanIf`, then `CanIf_CheckWakeup()` shall return `E_NOT_OK`. ⌋

[SWS_CANIF_00679] ⌈ If the `CCMSM` (see section 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`, `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` / `CAN Transceiver` detects a bus wake-up event, then this will be notified to the *ECU State Manager* directly. If such a *wake-up event* needs to be validated, the `EcuM` (or a *CDD*) switches on the corresponding `CAN Controller` (`CanIf_SetControllerMode()`) and `CAN Transceiver` (`CanIf_SetTrcvMode()`) (For more details see chapter 9 of [14]).

Attention: `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_ONLINE` or `CANIF_TX_ONLINE`. Thus, it is necessary that the *PDU Channel Mode* is not set to `CANIF_ONLINE` or `CANIF_TX_ONLINE` if wake-up validation is required.

Note: As per [SWS_CAN_00411] and *CAN Controller State Diagram* (see [1]) a direct transition from mode `CAN_T_SLEEP` to `CAN_T_START` is not allowed.

[SWS_CANIF_00226] ⌈ `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 *ECUC_CanIf_00637*) and by `CanDrv` configuration
- and/or underlying `CAN Transceiver` provides wake-up support and wake-up is enabled by the parameter `CANIF_TRCV_WAKEUP_SUPPORT` (see *ECUC_CanIf_00606*) and by `CanTrcv` configuration
- and configuration parameter `CANIF_PUBLIC_WAKEUP_CHECK_VALIDATION_SUPPORT` (see *ECUC_CanIf_00611*) is enabled.

⌋

[SWS_CANIF_00286] ⌈ If `CANIF_PUBLIC_WAKEUP_CHECK_VALIDATION_SUPPORT` equals `TRUE` `CanIf` enables the detection for *CAN wake-up validation*. Therefore, `CanIf` stores the event of the first called `CanIf_RxIndication()` of a `CAN Controller` which has been set to `CANIF_CS_STARTED`. ⌋

[SWS_CANIF_00179] ⌈ <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 CanIf at the corresponding CAN Controller. ⌋*(SRS_CAN_01136)*

Note: If there is no *wake-up event* stored in CanIf, CanIf_CheckValidation() should not call <User_ValidateWakeupEvent>().

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

- sources: EcuM_WakeupSourceType (see [14])

[SWS_CANIF_00756] ⌈ When CCMSM is set to CANIF_CS_SLEEP the stored event (first call of 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.

Figure 7.10 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 PduR or the Network Management are responsible for controlling the PDU operation modes.

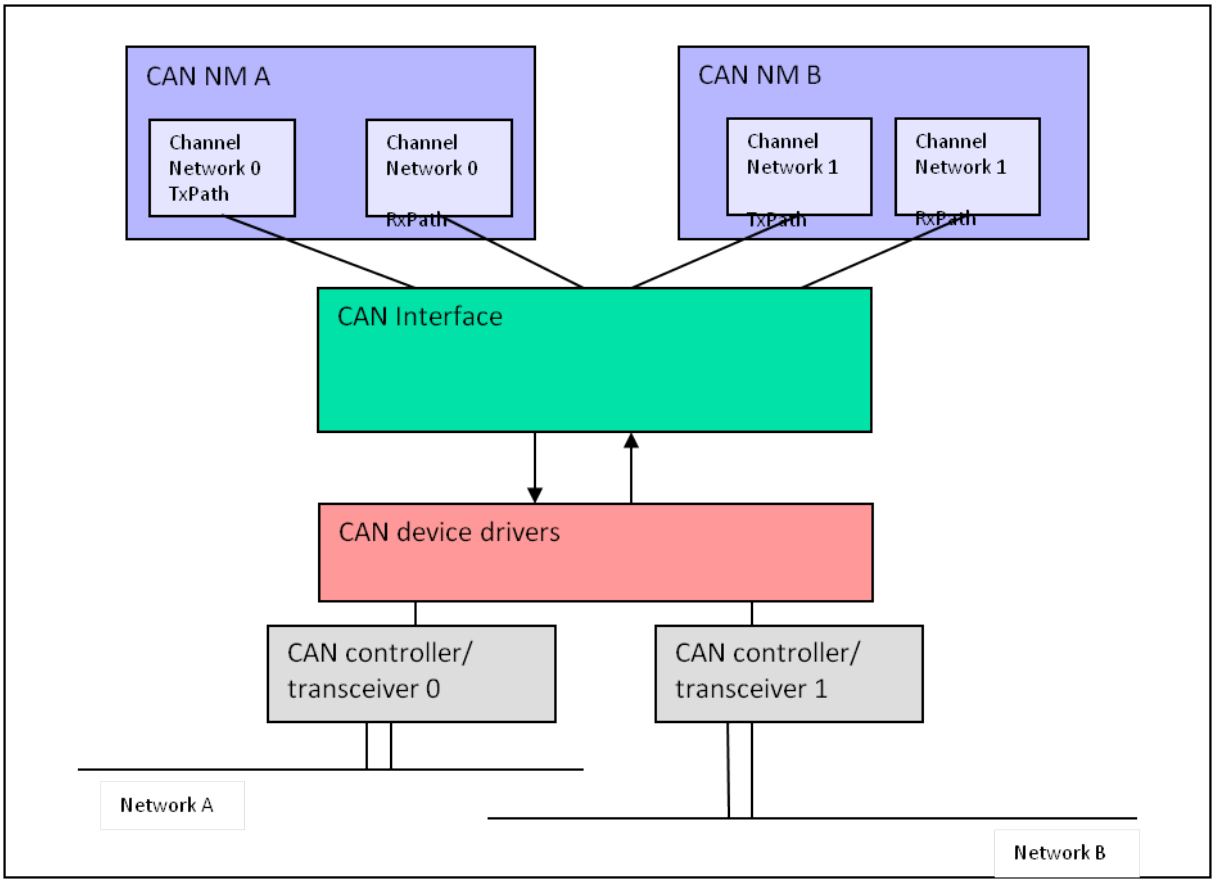

Figure 7.10: Channel PDU groups

7.20.2 PDU channel modes

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

- all Transmit L-PDUs belonging to one logical channel,
- all Transmit L-PDUs and Receive L-PDUs belonging to one logical channel.

Changing the PDU channel mode is only allowed during the network mode CANIF_CS_STARTED (refer to CANIF_CS_STARTED and [SWS_CANIF_00874]).

While CANIF_ONLINE and CANIF_OFFLINE affecting the whole communicatoin the PDU channel modes CANIF_TX_OFFLINE and CANIF_TX_OFFLINE_ACTIVE enable/disable transmission path seperately.

CanIf provides information about the current PDU channel mode via the service CanIf_GetPduMode().

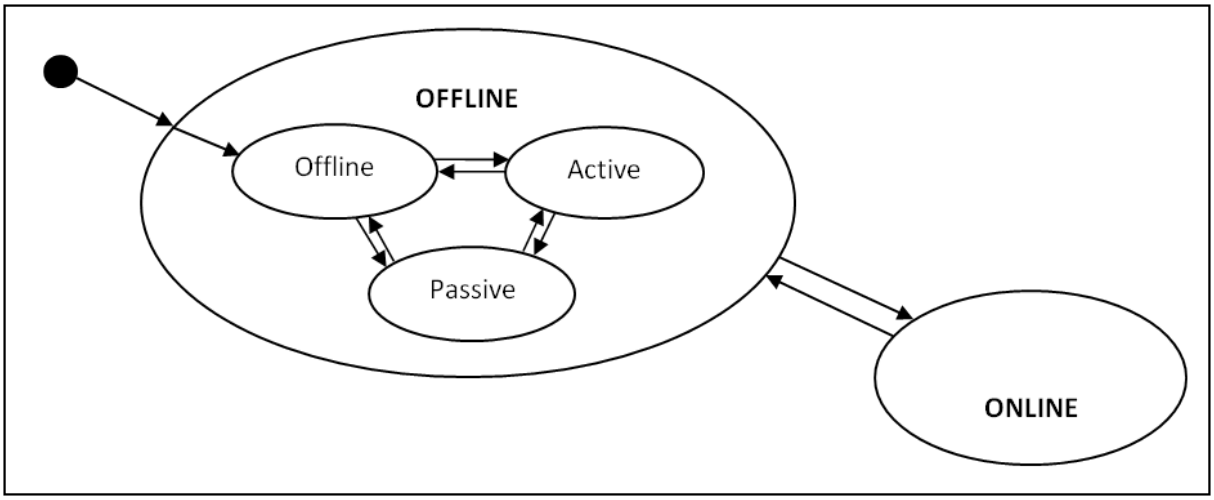

Figure 7.11: PDU channel mode control

Figure 7.11 shows a diagram with possible PDU channel modes. Each L-PDU channel can be in CANIF_OFFLINE (no communication), CANIF_TX_OFFLINE (passive mode => listen without sending), CANIF_TX_OFFLINE_ACTIVE (simulated transmission without listening (see [SWS_CANIF_00072]), and CANIF_ONLINE (full communication). The default state is the CANIF_OFFLINE mode.

7.20.2.1 CANIF_OFFLINE

[SWS_CANIF_00864] ⌈ During initialization CanIf shall switch every channel to CANIF_OFFLINE. ⌋

[SWS_CANIF_00865] ⌈ If CanIf_SetControllerMode(ControllerId, CANIF_CS_SLEEP) is called, CanIf shall set the PDU channel mode of the corresponding channel to CANIF_OFFLINE. ⌋

[SWS_CANIF_00073] ⌈ For Physical Channels switching to CANIF_OFFLINE mode CanIf shall:

- prevent forwarding of transmit requests CanIf_Transmit() of associated L-PDUs to CanDrv (return 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.

⌋

[SWS_CANIF_00866] ⌈ If `CanIf_SetControllerMode(ControllerId, CANIF_CS_STOPPED)` is called, `CanIf` shall set the PDU channel mode of the corresponding channel to `CANIF_TX_OFFLINE`. ⌋

[SWS_CANIF_00489] ⌈ For `Physical Channels` switching to `CANIF_TX_OFFLINE` mode `CanIf` shall:

- prevent forwarding of transmit requests `CanIf_Transmit()` of associated `L-PDUs` to `CanDrv` (return `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.
- enable invocation of receive indication callback services of the upper layer modules.

⌋

The *BusOff* notification is implicitly suppressed in case of `CANIF_OFFLINE` and `CANIF_TX_OFFLINE` due to the fact, that 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.

[SWS_CANIF_00118] ⌈ If those `Transmit L-PDUs`, which are already waiting for transmission in the `CAN Transmit Hardware Object`, will be transmitted immediately after change to `CANIF_TX_OFFLINE` or `CANIF_OFFLINE` mode and a subsequent *BusOff* event occurs, `CanIf` does not prohibit execution of the *BusOff* notification `<User_ControllerBusOff>(ControllerId)`. ⌋

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

7.20.2.2 CANIF_ONLINE

[SWS_CANIF_00075] ⌈ For `Physical Channels` switching to `CANIF_ONLINE` mode `CanIf` shall:

- enable forwarding of transmit requests `CanIf_Transmit()` of associated `L-PDUs` to `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.

⌋

7.20.2.3 CANIF_OFFLINE_ACTIVE

If CanIfTxOfflineActiveSupport = TRUE CanIf provides simulation of successful transmission by CANIF_TX_OFFLINE_ACTIVE mode. This mode is enabled by call of CanIf_SetPduMode(ControllerId, CANIF_TX_OFFLINE_ACTIVE) and only affects the transmission path.

[SWS_CANIF_00072] ⌈ For every L-PDU assigned to a channel which is in CANIF_TX_OFFLINE_ACTIVE mode CanIf shall call the transmit confirmation callback services of the upper layer modules immediately instead of buffering or forwarding of the L-PDUs to CanDrv during the call of CanIf_Transmit(). ⌋

Note: During CANIF_TX_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 [SWS_CANIF_00072]).

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. 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 CanId currently being processed. After the L-PDU Handle is found, CanIf accepts the L-PDU and enables upper layers to access L-SDU 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 where the CanId or the complete ID range is inside the defined range.

[SWS_CANIF_00645] ⌈ A range of CanIds which shall pass the software receive filter shall either be defined by its upper limit (see CANIF_HRHRANGE_UPPER_CANID, *ECUC_CanIf_00630*) and lower limit (see CANIF_HRHRANGE_LOWER_CANID, *ECUC_CanIf_00629*) CanId, or by a base ID (see CANIF_HRHRANGE_BASEID) and a mask that defines the relevant bits of the base ID (see CANIF_HRHRANGE_MASK). ⌋

Note: Software receive filtering is optional (see multiplicity of $0..*$ in *ECUC_CanIf_00628*).

[SWS_CANIF_00646] ⌈ Each configurable range of CanIds (see [SWS_CANIF_00645]), which shall pass the software receive filter, shall be configurable either for *Standard CAN IDs* or *Extended CAN IDs* via CANIF_HRHRANGE_CANIDTYPE (see *ECUC_CanIf_00644*). ⌋

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.

[SWS_CANIF_00030] ⌈ If 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 CanIf shall accept this L-PDU and the software filtering algorithm shall derive the Receive L-PDU Handle from the found CanId. ⌋*(SRS_CAN_01018)*

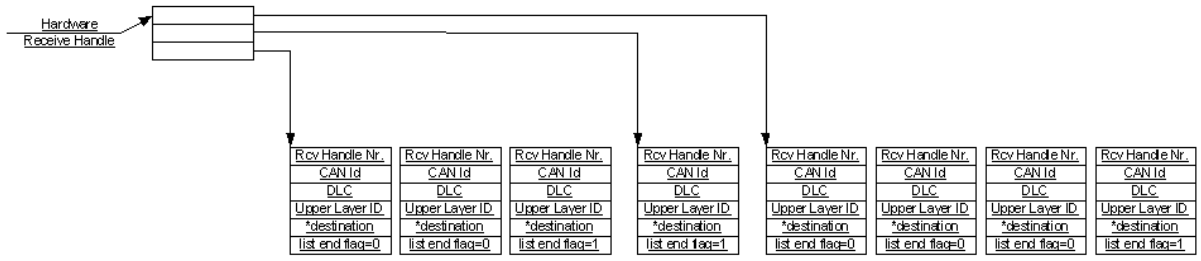

Figure 7.12: Software filtering example

[SWS_CANIF_00852] ⌈ If a range is (partly) contained in another range, or a single CanId is contained in a range, the software filter shall select the L-PDU Handle based on the following assumptions:

- A single CanId is always more relevant than a range.
- A smaller range is more relevant than a larger range.

⌋

7.21.2 Software filter algorithms

The choice of suitable software search algorithms it is up to the implementation of CanIf. 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.

[SWS_CANIF_00026] ⌈ CanIf shall accept all received L-PDUs (see [SWS_CANIF_00390]) with a DLC value equal or greater then the configured DLC value (see *ECUC_CanIf_00599*). ⌋*(SRS_CAN_01005)*

Hint: The DLC Check can be enabled or disabled globally by CanIf configuration (see parameter CANIF_PRIVATE_DLC_CHECK, *ECUC_CanIf_00617*) for all used CanDrvs.

[SWS_CANIF_00168] ⌈ If the DLC check rejects a received L-PDU (see [SWS_CANIF_00026]), CanIf shall report development error code CANIF_E_INVALID_DLC to the Det_ReportError() service of the DET module. ⌋

[SWS_CANIF_00829] ⌈ CanIf shall pass the received (see [SWS_CANIF_00006]) length value (DLC) to the target upper layer module (see [SWS_CANIF_00135]), if the DLC check is passed. ⌋

[SWS_CANIF_00830] ⌈ CanIf shall pass the received (see [SWS_CANIF_00006]) length value (DLC) to the target upper layer module (see [SWS_CANIF_00135]), if the DLC check is not configured (see *ECUC_CanIf_00617*) ⌋

7.23 L-SDU dispatcher to upper layers

Rationale: At transmission side the L-SDU dispatcher has to find out the corresponding Tx confirmation callback service of the target upper layer module. At reception side each L-SDU handle belongs to one single upper layer module as destination for the corresponding receive L-SDU or group of such L-SDUs. This relation is assigned statically at configuration time. The task of the L-SDU dispatcher inside of CanIf is to find out the customer for a received L-SDU 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 section 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 section 7.11 Transmit buffering).

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

7.25 Multiple CAN Driver support

`CanIf` needs a specific mapping to cover multiple `CanDrv` to provide a common interface to upper layers. Thus, `CanIf` must dispatch all actions up-down to the APIs of the corresponding `CanDrv` and underlying `CAN Controller`(s). For the way down-up `CanIf` has to provide adequate callback notifications to differentiate between multiple `CanDrvs`.

The support for multiple `CanDrvs` can be enabled and disabled by the configuration parameter `CANIF_MULTIPLE_DRIVER_SUPPORT` (see *ECUC_CanIf_00612*).

7.25.1 Transmit requests by using multiple CAN Drivers

Each `Transmit L-PDU` enables `CanIf` to derive the corresponding `CAN Controller` and implicitly `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 CanIf should be provided. 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 different CanDrvs. For an example refer to subsection 7.25.3 Mapping table for multiple CAN Driver handling.

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: The Figure 7.13 and the following table serve only as an example. Finally, it is up to the implementation to access the correct APIs of underlying CanDrvs.

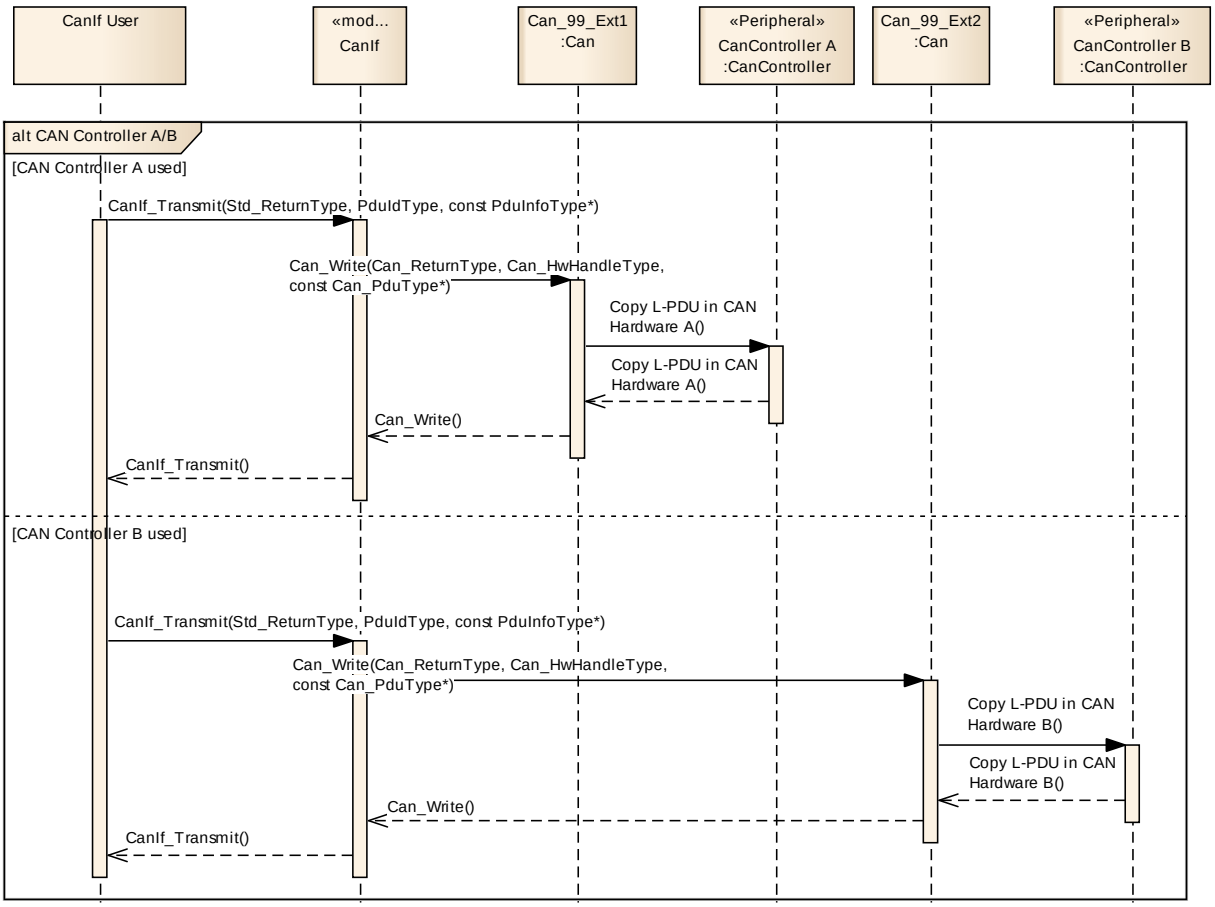

Figure 7.13: Transmission request with multiple CAN Drivers - simplified

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 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 (Hth, PduInfoPtr)	Request for transmission to the corresponding CAN_Driver serving i.e. CAN Controller #0 within the "A" Hardware Unit.
Hardware request	All L-PDU data will be set in the 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 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 (Hth, PduInfoPtr_2)	Request for transmission to the corresponding CAN_Driver 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. CAN Controller #1 within Hardware Unit "B" and the transmit request enabled.

7.25.2 Notification mechanism using multiple CAN Drivers

Even if multiple CanDrvs are used in a single ECU Every notification callback service invoked by CanDrvs at the CanIf exists only once. This means, that CanIf has to identify calling CanDrv using the passed parameters. Thus, there has to be at least one parameter which clearly specifies calling CanDrv.

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

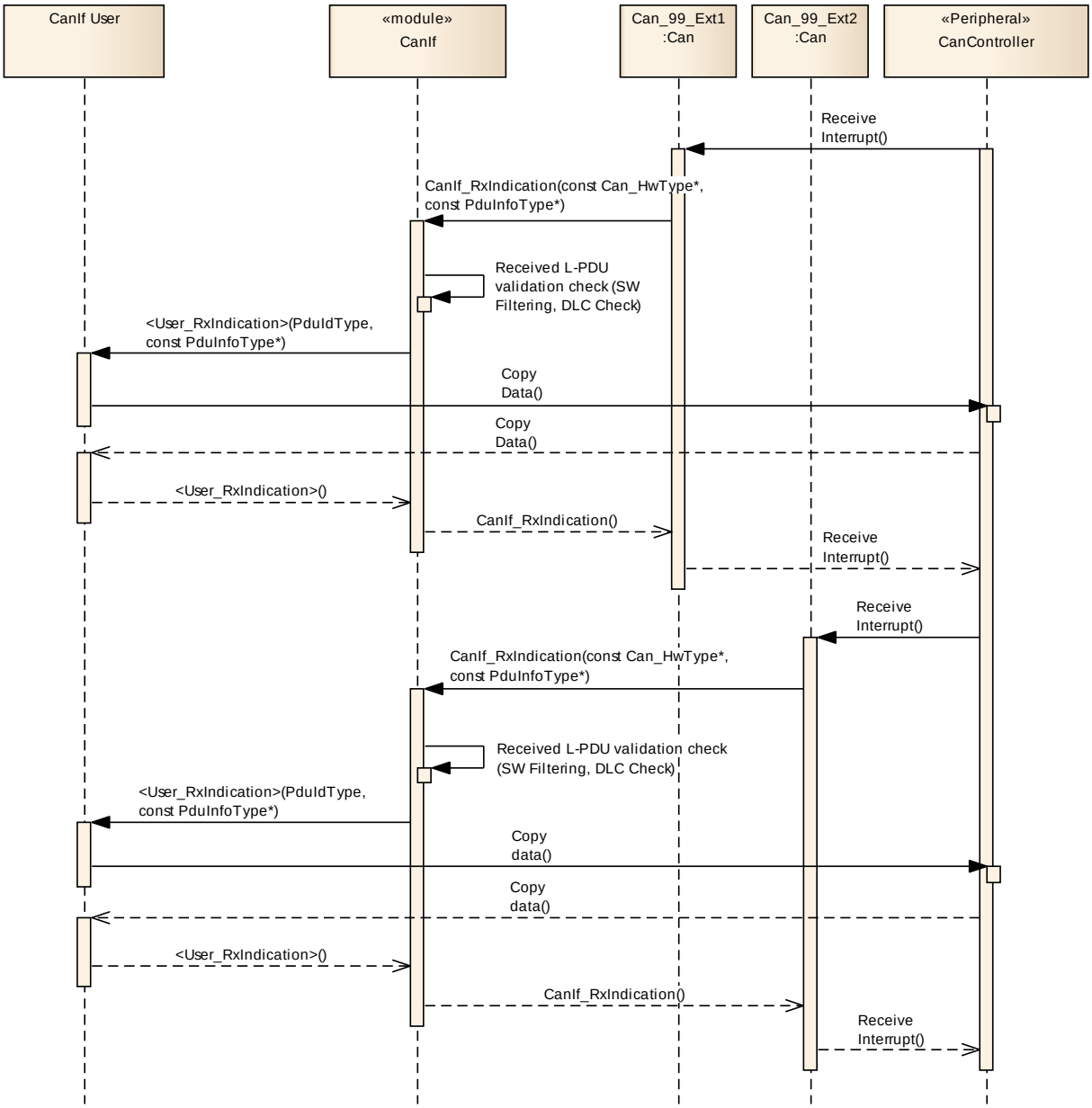

Figure 7.14: Receive interrupt with multiple `CanDrvs` - simplified

Operations called	Description
Receive Interrupt	`CAN Controller` 1 signals a successful reception and triggers a *receive interrupt*. The *ISR* of `CanDrv` A is invoked.
`CanIf_RxIndication (Mailbox_1, PduInfoPtr_1)`	The reception is indicated to `CanIf` by calling of `CanIf_RxIndication()`. The pointer `Mailbox_1` identifies the `HRH` and its corresponding `CAN Controller`, which contains the received `L-PDU` specified by `PduInfoPtr_1`.
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-SDU` is not indicated to upper layers and 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_1, CanPduInfoPtr_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_1 specifies the ID of the received L-SDU. The second parameter is the reference on PduInfoType which provides access to the buffer containing 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 (Mailbox_2, PduInfoPtr_2)	The reception is indicated to CanIf by calling of CanIf_RxIndication(). The pointer Mailbox_2 identifies the HRH and its corresponding CAN Controller, which contains the received L-PDU specified by PduInfoPtr_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-SDU is not indicated to upper layers and 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, CanPduInfoPtr_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 ID of the received L-SDU. The second parameter is the reference on PduInfoType which provides access to the buffer containing 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 CanIf. This table makes the assignment from two different driver interfaces to one single driver interface.

In case of L-SDU Handle based APIs, CanIf has to derive the corresponding CanDrv from the L-SDU Handle. Afterwards CanIf can use the *CanDrv Number* as an index for the table with function pointers. The parameters have to be translated accordingly: i.e. L-SDU Handle => HTH/HRH, CanId, *DLC*.

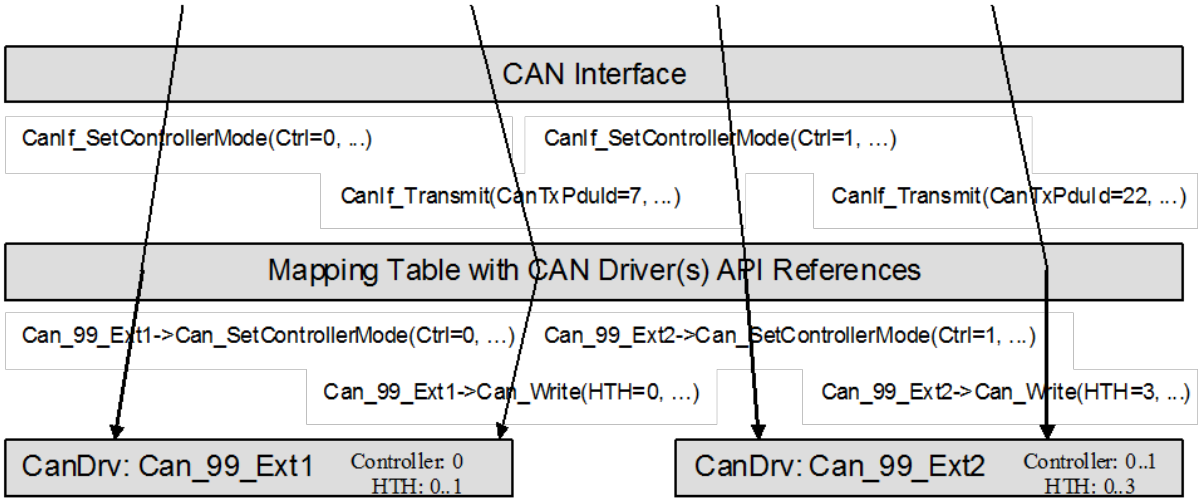

Figure 7.15: HTH Assignment with multiple CAN Drivers

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 Controllers and HTHs.

7.26 Partial Networking

[SWS_CANIF_00747] ⌈ If *Partial Networking* (PN) is enabled (see CANIF_PUBLIC_PN_SUPPORT, *ECUC_CanIf_00772*), CanIf shall support a PnTxFilter per CAN Controller which overlays the *PDU channel modes*. ⌋

[SWS_CANIF_00748] ⌈ The PnTxFilter of [SWS_CANIF_00747] shall only have an effect and transition its modes (enabled/disabled) if more than zero Tx L-PDUs per CAN Controller are configured as CanIfTxPduPnFilterPdu (see CANIF_TXPDU_PNFILTERPDU, *ECUC_CanIf_00773*). ⌋

[SWS_CANIF_00863] ⌈ PnTxFilter shall be enabled during initialization (ref. to [SWS_CANIF_00747] and [SWS_CANIF_00748]). ⌋

[SWS_CANIF_00749] ⌈ If CanIf_SetControllerMode(ControllerId, CANIF_CS_SLEEP) is called the PnTxFilter of the corresponding CAN Controller shall be enabled (ref. to [SWS_CANIF_00748] and [SWS_CANIF_00747]). ⌋

[SWS_CANIF_00750] ⌈ If the PnTxFilter of a CAN Controller is enabled, CanIf shall block all Tx requests to that CAN Controller (return E_NOT_OK when CanIf_Transmit() is called), except if the requested Tx L-PDUs is one of the configured CanIfTxPduPnFilterPdus of that CAN Controller. These CanIfTxPduPnFilterPdus shall always be passed to the corresponding CAN Driver. ⌋

[SWS_CANIF_00751] ⌈ If `CanIf_TxConfirmation()` is called, the corresponding `PnTxFilter` shall be disabled (ref. to [SWS_CANIF_00747] and [SWS_CANIF_00748]). ⌋

[SWS_CANIF_00752] ⌈ If the `PnTxFilter` of a `CAN Controller` is disabled, `CanIf` shall behave as requested via `CanIf_SetPduMode` (see [SWS_CANIF_00008]). ⌋

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.

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_HOH 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 CANIF_E_PARAM_PDU_MODE	10 12 13 14 15 16 17 18 19 21 22
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 mode state machine is not in mode CANIF_CS_SLEEP	Development	CANIF_E_NOT_SLEEP	71

7.28 Error detection

[SWS_CANIF_00661] ⌈ 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

[SWS_CANIF_00223] ⌈ 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. ⌋

[SWS_CANIF_00119] ⌈ 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. ⌋

8 API specification

8.1 Imported types

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

[SWS_CANIF_00142] ⌈

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

Table 8.1: CanIf_ImportedTypes

⌋(*SRS_BSW_00348*, *SRS_BSW_00353*, *SRS_BSW_00361*)

8.2 Type definitions

8.2.1 CanIf_ConfigType

[SWS_CANIF_00144] ⌈

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.

Table 8.2: CanIf_ConfigType

⌋

[SWS_CANIF_00523] ⌈ 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/L-SDU handle. ⌋

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

Note: The definition of CAN Interface public parameters contains:

- Number of transmit L-PDUs/L-SDUs
- Number of receive L-PDUs/L-SDUs
- Number of dynamic transmit L-PDU/L-SDU handles

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

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

8.2.2 CanIf_ControllerModeType

[SWS_CANIF_00136] ⌈

Name:	**CanIf_ControllerModeType**
Type:	Enumeration

Range:	CANIF_CS_UNINIT	UNINIT mode. Default mode of each CAN controller after power on.
	CANIF_CS_SLEEP	The CAN controller is in SLEEP mode and can be woken up by an internal (SW) request or by a network event (This must be supported by CAN hardware.).
	CANIF_CS_STARTED	The CAN controller is in full-operational mode.
	CANIF_CS_STOPPED	The CAN controller is halted and does not operate on the network.
Description:	Operating modes of a CAN controller.	

Table 8.3: CanIf_ControllerModeType

⌋

8.2.3 CanIf_PduModeType

[SWS_CANIF_00137] ⌈

Name:	**CanIf_PduModeType**	
Type:	Enumeration	
Range:	CANIF_OFFLINE	= 0 Transmit and receive path of the corresponding channel are disabled => no communication mode
	CANIF_TX_OFFLINE	Transmit path of the corresponding channel is disabled. The receive path is enabled.
	CANIF_TX_OFFLINE_ACTIVE	Transmit path of the corresponding channel is in offline active mode (see SWS_CANIF_00072). The receive path is disabled. This mode requires CanIfTxOfflineActiveSupport = TRUE.
	CANIF_ONLINE	Transmit and receive path of the corresponding channel are enabled => full operation mode
Description:	The PduMode of a 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.	

Table 8.4: CanIf_PduModeType

⌋

8.2.4 CanIf_NotifStatusType

[SWS_CANIF_00201] ⌈

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

Table 8.5: CanIf_NotifStatusType

⌋

8.3 Function definitions

8.3.1 CanIf_Init

[SWS_CANIF_00001] Initialization interface ⌈

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 (in-out):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service Initializes internal and external interfaces of the CAN Interface for the further processing.	

Table 8.6: CanIf_Init

⌋(*SRS_BSW_00405*, *SRS_BSW_00101*, *SRS_BSW_00358*, *SRS_BSW_00414*, *SRS_CAN_01021*, *SRS_CAN_01022*)

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

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

[SWS_CANIF_00085] ⌈ 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 (subsection 8.2.1 CanIf_ConfigType) of chapter 10 Configuration specification are specified, they shall be used for initialization.

[SWS_CANIF_00302] ⌈ 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. ⌋(*SRS_BSW_00323*)

8.3.2 CanIf_SetControllerMode

[SWS_CANIF_00003] ⌈

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.	

Table 8.7: CanIf_SetControllerMode

⌋(*SRS_CAN_01027*)

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`.

[SWS_CANIF_00308] ⌈ The service `CanIf_SetControllerMode()` shall call `Can_SetControllerMode(Controller, Transition)` for the requested CAN controller. ⌋

[SWS_CANIF_00311] ⌈ 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. ⌋(*SRS_BSW_00323*)

[SWS_CANIF_00774] ⌈ 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. ⌋(*SRS_BSW_00323*)

[SWS_CANIF_00312] ⌈ 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

[SWS_CANIF_00229] ⌈

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.

Parameters (in-out):	None	
Parameters (out):	ControllerMode Ptr	Pointer to a memory location, where the current mode of the CAN controller will be stored.
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.	

Table 8.8: CanIf_GetControllerMode

⌋(*SRS_CAN_01028*)

[SWS_CANIF_00541] ⌈ 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 subsection 7.19.2). ⌋

[SWS_CANIF_00313] ⌈ 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. ⌋(*SRS_BSW_00323*)

[SWS_CANIF_00656] ⌈ 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. ⌋(*SRS_BSW_00323*)

[SWS_CANIF_00316] ⌈ 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

[SWS_CANIF_00005] ⌈

Service name:	CanIf_Transmit
Syntax:	Std_ReturnType CanIf_Transmit(PduIdType CanIfTxSduId, const PduInfoType* CanIfTxInfoPtr)

Service ID[hex]:	0x05	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	CanIfTxSduId	L-SDU handle to be transmitted. This handle specifies the corresponding CAN L-SDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device.
	CanIfTxInfoPtr	Pointer to a structure with CAN L-SDU related data: DLC and pointer to CAN L-SDU buffer including the MetaData of dynamic L-PDUs.
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 CanTxSduId and CAN related data in the L-SDU structure.	

Table 8.9: CanIf_Transmit

⌋(*SRS_CAN_01008*)

Note: The corresponding CAN Controller and HTH have to be resolved by the CanIfTxSduId.

[SWS_CANIF_00317] ⌈ 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. ⌋

[SWS_CANIF_00318] ⌈ The service CanIf_Transmit() shall map

- the parameters of the data structure, the L-SDU handle with the identifier CanIfTxSduId refers to (*CanID*, HTH/HRH of the CAN Controller)
- and the pointer PduInfoPtr points to (*DLC*, pointer to L-SDU buffer),

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

Note: CanIfTxInfoPtr is a pointer to a L-SDU user memory, *CAN Identifier*, L-SDU handle and *DLC* (see [1, Specification of CAN Driver]).

[SWS_CANIF_00243] ⌈ CanIf shall set the two most significant bits ('IDentifier Extension flag' (see [13, ISO11898 (CAN)]) and 'CAN FD flag') of the *CanId* (CanIfTxInfoPtr->id) before CanIf passes the predefined *CanId* to CanDrv at call of Can_Write() (see [1, Specification of CAN Driver], definition of Can_IdType

[SWS_Can_00416]). The *CanId* format type of each CAN L-PDU can be configured by CANIF_TXPDU_CANIDTYPE, refer to *ECUC_CanIf_00590*. ⌋(*SRS_CAN_01141*)

[SWS_CANIF_00162] ⌈ 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.

[SWS_CANIF_00319] ⌈ If parameter CanIfTxSduId of CanIf_Transmit() has an invalid value, CanIf shall report development error code CANIF_E_INVALID_TXPDUID to the Det_ReportError service of the DET, when CanIf_Transmit() is called. ⌋(*SRS_BSW_00323*)

[SWS_CANIF_00320] ⌈ If parameter CanIfTxInfoPtr of CanIf_Transmit() has an invalid value, CanIf shall report development error code CANIF_E_PARAM_POINTER to the Det_ReportError service of the DET module, when CanIf_Transmit() is called. ⌋(*SRS_BSW_00323*)

[SWS_CANIF_00323] ⌈ Caveats of CanIf_Transmit():

- During the call of this API the buffer of CanIfTxInfoPtr is controlled by 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.
- CanIf must be initialized after Power ON.

⌋

8.3.5 CanIf_CancelTransmit

[SWS_CANIF_00520] ⌈

Service name:	CanIf_CancelTransmit	
Syntax:	Std_ReturnType CanIf_CancelTransmit(PduIdType CanIfTxSduId)	
Service ID[hex]:	0x18	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	CanIfTxSduId	L-SDU handle to be transmitted. This handle specifies the corresponding CAN L-SDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device.
Parameters (in-out):	None	

Parameters (out):	None	
Return value:	Std_ReturnType	Always return E_OK
Description:	This is a dummy method introduced for interface compatibility.	

Table 8.10: CanIf_CancelTransmit

⌋

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

[SWS_CANIF_00521] ⌈ The service `CanIf_CancelTransmit()` shall be pre-compile time configurable `On/Off` by the configuration parameter `CANIF_PUBLIC_CANCEL_TRANSMIT_SUPPORT` (see *ECUC_CanIf_00614*). It shall be configured `ON` if `PduRComCancelTransmitSupport` is configured as `ON`. ⌋

[SWS_CANIF_00652] ⌈ If parameter `CanIfTxSduId` of `CanIf_CancelTransmit()` has an invalid value, `CanIf` shall report development error code `CANIF_E_INVALID_TXPDUID` to the `Det_ReportError service` of the DET, when `CanIf_CancelTransmit()` is called. ⌋*(SRS_BSW_00323)*

8.3.6 CanIf_ReadRxPduData

[SWS_CANIF_00194] ⌈

Service name:	CanIf_ReadRxPduData	
Syntax:	Std_ReturnType CanIf_ReadRxPduData(PduIdType CanIfRxSduId, PduInfoType* CanIfRxInfoPtr)	
Service ID[hex]:	0x06	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	CanIfRxSduId	Receive L-SDU handle specifying the corresponding CAN L-SDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device.
Parameters (inout):	None	
Parameters (out):	CanIfRxInfoPtr	Pointer to a structure with CAN L-SDU related data: DLC and pointer to CAN L-SDU buffer including the MetaData of dynamic L-PDUs.

Return value:	Std_ReturnType	E_OK: Request for L-SDU 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 CanIfRxSduId to the calling upper layer.	

Table 8.11: CanIf_ReadRxPduData

⌋(*SRS_CAN_01125*, *SRS_CAN_01129*)

[SWS_CANIF_00324] ⌈ 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. ⌋

[SWS_CANIF_00325] ⌈ If parameter `CanIfRxSduId` of `CanIf_ReadRxPduData()` has an invalid value, e.g. not configured to be stored within `CanIf` via `CANIF_READRXPDU_DATA` (*ECUC_CanIf_00600*), `CanIf` shall report development error code `CANIF_E_INVALID_RXPDUID` to the `Det_ReportError service` of the DET, when `CanIf_ReadRxPduData()` is called. ⌋(*SRS_BSW_00323*)

[SWS_CANIF_00326] ⌈ If parameter `CanIfRxInfoPtr` of `CanIf_ReadRxPduData()` has an invalid value, `CanIf` shall report development error code `CANIF_E_PARAM_POINTER` to the `Det_ReportError service` of the DET module, when `CanIf_ReadRxPduData()` is called. ⌋(*SRS_BSW_00323*)

[SWS_CANIF_00329] ⌈ Caveats of `CanIf_ReadRxPduData()`:

- During the call of this API the buffer of `CanIfRxInfoPtr` is controlled by `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 `CanIfRxSduId`, which are defined to receive multiple CAN-Ids (range reception).
- `CanIf` must be initialized after Power ON.

⌋

[SWS_CANIF_00330] ⌈ 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` (*ECUC_CanIf_00607*). ⌋

8.3.7 CanIf_ReadTxNotifStatus

[SWS_CANIF_00202] ⌈

Service name:	CanIf_ReadTxNotifStatus

Syntax:	CanIf_NotifStatusType CanIf_ReadTxNotifStatus(PduIdType CanIfTxSduId)	
Service ID[hex]:	0x07	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	CanIfTxSduId	L-SDU handle to be transmitted. This handle specifies the corresponding CAN L-SDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device.
Parameters (in-out):	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 occurred or not) of a specific static or dynamic CAN Tx L-PDU, requested by the CanIfTxSduId.	

Table 8.12: CanIf_ReadTxNotifStatus

⌋(*SRS_CAN_01130*)

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

[SWS_CANIF_00393] ⌈ If configuration parameters CANIF_PUBLIC_READTXPDU_NOTIFY_STATUS_API (*ECUC_CanIf_00609*) and CANIF_TXPDU_READ_NOTIFYSTATUS (*ECUC_CanIf_00589*) for the transmitted L-SDU are set to TRUE, and if CanIf_ReadTxNotifStatus() is called, the CanIf shall reset the notification status for the transmitted L-SDU. ⌋

[SWS_CANIF_00331] ⌈ If parameter CanIfTxSduId of CanIf_ReadTxNotifStatus() is out of range or if no status information was configured for this CAN Tx L-SDU, CanIf shall report development error code CANIF_E_INVALID_TXPDUID to the Det_ReportError service of the DET when CanIf_ReadTxNotifStatus() is called. ⌋(*SRS_BSW_00323*)

[SWS_CANIF_00334] ⌈ Caveats of CanIf_ReadTxNotifyStatus(): CanIf must be initialized after Power ON. ⌋

[SWS_CANIF_00335] ⌈ 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 *ECUC_CanIf_00609*). ⌋

8.3.8 CanIf_ReadRxNotifStatus

[SWS_CANIF_00230] ⌈

Service name:	CanIf_ReadRxNotifStatus	
Syntax:	CanIf_NotifStatusType CanIf_ReadRxNotifStatus(PduIdType CanIfRxSduId)	
Service ID[hex]:	0x08	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	CanIfRxSduId	Receive L-SDU handle specifying the corresponding CAN L-SDU 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 CanIfRxSduId.	

Table 8.13: CanIf_ReadRxNotifStatus

⌋(*SRS_CAN_01130*, *SRS_CAN_01131*)

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

[SWS_CANIF_00394] ⌈ If configuration parameters CANIF_PUBLIC_READRXPDU_NOTIFY_STATUS_API (*ECUC_CanIf_00608*) and CANIF_RXPDU_READ_NOTIFYSTATUS (*ECUC_CanIf_00595*) are set to TRUE, and if CanIf_ReadRxNotifStatus() is called, then CanIf shall reset the notification status for the received L-SDU. ⌋

[SWS_CANIF_00336] ⌈ If parameter CanIfRxSduId 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-SDU, CanIf shall report development error code CANIF_E_INVALID_RXPDUID to the Det_ReportError service of the DET, when CanIf_ReadRxNotifStatus() is called. ⌋(*SRS_BSW_00323*)

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

[SWS_CANIF_00339] ⌈ Caveats of `CanIf_ReadRxNotifStatus()`:

- `CanIf` must be initialized after Power ON.

⌋

[SWS_CANIF_00340] ⌈ 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 *ECUC_CanIf_00608*). ⌋

8.3.9 CanIf_SetPduMode

[SWS_CANIF_00008] ⌈

Service name:	CanIf_SetPduMode	
Syntax:	Std_ReturnType CanIf_SetPduMode(uint8 ControllerId, CanIf_PduModeType 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
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.	

Table 8.14: CanIf_SetPduMode

⌋

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

[SWS_CANIF_00341] ⌈ If `CanIf_SetPduMode()` is called with invalid `ControllerId`, `CanIf` shall report development error code

CANIF_E_PARAM_CONTROLLERID to the Det_ReportError service of the DET module. ⌋(*SRS_BSW_00323*)

[SWS_CANIF_00860] ⌈ If CanIf_SetPduMode() is called with invalid PduModeRequest, CanIf shall report development error code CANIF_E_PARAM_PDU_MODE to the Det_ReportError service of the *DET* module. ⌋(*SRS_BSW_00323*)

[SWS_CANIF_00874] ⌈ The service CanIf_SetPduMode() shall not accept any request and shall return E_NOT_OK, if the CCMSM referenced by ControllerId is not in state CANIF_CS_STARTED. ⌋

[SWS_CANIF_00344] ⌈ Caveats of CanIf_SetPduMode(): CanIf must be initialized after Power ON. ⌋

8.3.10 CanIf_GetPduMode

[SWS_CANIF_00009] ⌈

Service name:	CanIf_GetPduMode	
Syntax:	Std_ReturnType CanIf_GetPduMode(uint8 ControllerId, CanIf_PduModeType* 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.	

Table 8.15: CanIf_GetPduMode

⌋

[SWS_CANIF_00346] ⌈ If CanIf_GetPduMode() is called with invalid ControllerId, CanIf shall report development error code

CANIF_E_PARAM_CONTROLLERID to the Det_ReportError service of the DET module. ⌋(*SRS_BSW_00323*)

[SWS_CANIF_00657] ⌈ If CanIf_GetPduMode() is called with invalid PduModePtr, CanIf shall report development error code CANIF_E_PARAM_POINTER to the Det_ReportError service of the DET module. ⌋(*SRS_BSW_00323*)

[SWS_CANIF_00349] ⌈ Caveats of CanIf_GetPduMode(): CanIf must be initialized after Power ON. ⌋

8.3.11 CanIf_GetVersionInfo

[SWS_CANIF_00158] ⌈

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.	

Table 8.16: CanIf_GetVersionInfo

⌋(*SRS_BSW_00407*, *SRS_BSW_00411*)

8.3.12 CanIf_SetDynamicTxId

[SWS_CANIF_00189] ⌈

Service name:	CanIf_SetDynamicTxId
Syntax:	void CanIf_SetDynamicTxId(PduIdType CanIfTxSduId, Can_IdType CanId)
Service ID[hex]:	0x0c

Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	CanIfTxSduId	L-SDU handle to be transmitted. This handle specifies the corresponding CAN L-SDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device.
	CanId	Standard/Extended CAN ID of CAN L-SDU that shall be transmitted as FD or conventional CAN frame.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service reconfigures the corresponding CAN identifier of the requested CAN L-PDU.	

Table 8.17: CanIf_SetDynamicTxId

⌋

[SWS_CANIF_00352] ⌈ If parameter `CanIfTxSduId` of `CanIf_SetDynamicTxId()` has an invalid value, `CanIf` shall report development error code `CANIF_E_INVALID_TXPDUID` to the `Det_ReportError service` of the DET module, when `CanIf_SetDynamicTxId()` is called. ⌋*(SRS_BSW_00323)*

[SWS_CANIF_00353] ⌈ If parameter `CanId` of `CanIf_SetDynamicTxId()` has an invalid value, `CanIf` shall report development error code `CANIF_E_PARAM_CANID` to the `Det_ReportError service` of the DET module, when `CanIf_SetDynamicTxId()` is called. ⌋*(SRS_BSW_00323)*

[SWS_CANIF_00355] ⌈ If `CanIf` was not initialized before calling `CanIf_SetDynamicTxId()`, then the function `CanIf_SetDynamicTxId()` shall not execute a reconfiguration of Tx CanId. ⌋

[SWS_CANIF_00356] ⌈ Caveats of `CanIf_SetDynamicTxId()`:

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

⌋

[SWS_CANIF_00357] ⌈ Configuration of `CanIf_SetDynamicTxId()`: This function shall be pre compile time configurable `On/Off` by the configuration parameter `CANIF_PUBLIC_SETDYNAMICTXID_API` (see *ECUC_CanIf_00610*). ⌋

8.3.13 CanIf_SetTrcvMode

[SWS_CANIF_00287] ⌈

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 (in-out):	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.	

Table 8.18: CanIf_SetTrcvMode

⌋

Note: For more details, please refer to the [2, Specification of CAN Transceiver Driver].

[SWS_CANIF_00358] ⌈ 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 [2, Specification of CAN Transceiver Driver])

[SWS_CANIF_00538] ⌈ 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. ⌋*(SRS_BSW_00323)*

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 [2]). 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 [2]). But this is not checked by the CanIf.

[SWS_CANIF_00648] ⌈ 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. ⌋(*SRS_BSW_00323*)

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.

[SWS_CANIF_00362] ⌈ Configuration of CanIf_SetTrcvMode(): The number of supported transceiver types for each network is set up in the configuration phase (see CanInterfaceTransceiverConfiguration *ECUC_CanIf_00587* and CanInterfaceTransceiverDriverConfiguration *ECUC_CanIf_00273*). If no transceiver is used, this function shall not be provided. ⌋

8.3.14 CanIf_GetTrcvMode

[SWS_CANIF_00288] ⌈

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 (in-out):	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.	

Table 8.19: CanIf_GetTrcvMode

⌋

Note: For more details, please refer to the [2, Specification of CAN Transceiver Driver].

[SWS_CANIF_00363] ⌈ 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 [2, Specification of CAN Transceiver Driver])

[SWS_CANIF_00364] ⌈ 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. ⌋*(SRS_BSW_00323)*

[SWS_CANIF_00650] ⌈ 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. ⌋*(SRS_BSW_00323)*

[SWS_CANIF_00367] ⌈ Configuration of `CanIf_GetTrcvMode()`: The number of supported transceiver types for each network is set up in the configuration phase (see CanInterfaceTransceiverConfiguration *ECUC_CanIf_00587* and CanInterfaceTransceiverDriverConfiguration *ECUC_CanIf_00273*). If no transceiver is used, this function shall not be provided. ⌋

8.3.15 CanIf_GetTrcvWakeupReason

[SWS_CANIF_00289] ⌈

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 (in-out):	None	
Parameters (out):	TrcvWuReason Ptr	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.	

Table 8.20: CanIf_GetTrcvWakeupReason

⌋

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 [2, Specification of CAN Transceiver Driver].

[SWS_CANIF_00368] ⌈ 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 [2, Specification of CAN Transceiver Driver])

[SWS_CANIF_00537] ⌈ 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. ⌋*(SRS_BSW_00323)*

[SWS_CANIF_00649] ⌈ 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. ⌋*(SRS_BSW_00323)*

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 unpowered. 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.

[SWS_CANIF_00371] ⌈ Configuration of `CanIf_GetTrcvWakeupReason()`: The number of supported transceiver types for each network is set up in the configuration phase (see CanInterfaceTransceiverConfiguration *ECUC_CanIf_00587* and CanInterfaceTransceiverDriverConfiguration *ECUC_CanIf_00273*). If no transceiver is used, this function shall not be provided. ⌋

8.3.16 CanIf_SetTrcvWakeupMode

[SWS_CANIF_00290] ⌈

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.
	TrcvWakeup Mode	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.	

Table 8.21: CanIf_SetTrcvWakeupMode

⌋

Note: For more details, please refer to [2, Specification of CAN Transceiver Driver].

[SWS_CANIF_00372] ⌈ 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 [2, Specification of CAN Transceiver Driver])
- `Transceiver`: `uint8` (Transceiver to which API call has to be applied)

(see [2, Specification of CAN Transceiver Driver])

Note: The following three paragraphs are already described in the Specification of CanTrcv (see [2]). 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`.

[SWS_CANIF_00535] ⌈ 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. ⌋*(SRS_BSW_00323)*

[SWS_CANIF_00536] ⌈ 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. ⌋*(SRS_BSW_00323)*

[SWS_CANIF_00373] ⌈ Configuration of `CanIf_SetTrcvWakeupMode()`: The number of supported transceiver types for each network is set up in the configuration phase (see CanInterfaceTransceiverConfiguration *ECUC_CanIf_00587* and CanInterfaceTransceiverDriverConfiguration *ECUC_CanIf_00273*). If no transceiver is used, this function shall not be provided. ⌋

8.3.17 CanIf_CheckWakeup

[SWS_CANIF_00219] ⌈

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.	

Table 8.22: CanIf_CheckWakeup

⌋

Note: *Integration Code* calls this function

[SWS_CANIF_00398] ⌈ 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. ⌋*(SRS_BSW_00323)*

[SWS_CANIF_00401] ⌈ 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.

⌋

[SWS_CANIF_00402] ⌈ Configuration of `CanIf_CheckWakeup()`: This wake-up service is configurable by `CANIF_CTRL_WAKEUP_SUPPORT` (see *ECUC_CanIf_00637*) and/or `CANIF_TRCV_WAKEUP_SUPPORT` (see *ECUC_CanIf_00606*), 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

[SWS_CANIF_00178] ⌈

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 (in-out):	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.	

Table 8.23: CanIf_CheckValidation

⌋

Note: *Integration Code* calls this function

[SWS_CANIF_00404] ⌈ 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. ⌋*(SRS_BSW_00323)*

[SWS_CANIF_00407] ⌈ 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.

⌋

[SWS_CANIF_00408] ⌈ Configuration of `CanIf_CheckValidation()`: If no validation is needed, this API can be omitted by disabling of `CANIF_PUBLIC_WAKEUP_CHECK_VALIDATION_SUPPORT` (see *ECUC_CanIf_00611*). ⌋

8.3.19 CanIf_GetTxConfirmationState

[SWS_CANIF_00734] ⌈

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 (in-out):	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.

Table 8.24: CanIf_GetTxConfirmationState

⌋

[SWS_CANIF_00736] ⌈ 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. ⌋

[SWS_CANIF_00737] ⌈ Caveats of `CanIf_GetTxConfirmationState()`:

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

⌋

[SWS_CANIF_00738] ⌈ Configuration of `CanIf_GetTxConfirmationState()`: If BusOff Recovery of CanSm doesn't need the status of the Tx confirmations (see [SWS_CANIF_00740]), this API can be omitted by disabling of `CANIF_PUBLIC_TXCONFIRM_POLLING_SUPPORT` (see *ECUC_CanIf_00733*). ⌋

8.3.20 CanIf_ClearTrcvWufFlag

[SWS_CANIF_00760] ⌈

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 (in-out):	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.

Table 8.25: CanIf_ClearTrcvWufFlag

⌋

[SWS_CANIF_00766] ⌈ Within `CanIf_ClearTrcvWufFlag()` the function `CanTrcv_ClearTrcvWufFlag()` shall be called. ⌋

[SWS_CANIF_00769] ⌈ 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. ⌋

[SWS_CANIF_00771] ⌈ 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 *ECUC_CanIf_00772*). ⌋

8.3.21 CanIf_CheckTrcvWakeFlag

[SWS_CANIF_00761] ⌈

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.	

Table 8.26: CanIf_CheckTrcvWakeFlag

⌋

[SWS_CANIF_00765] ⌈ Within `CanIf_CheckTrcvWakeFlag()` the function `CanTrcv_CheckTrcvWakeFlag()` shall be called. ⌋

[SWS_CANIF_00770] ⌈ 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. ⌋

[SWS_CANIF_00813] ⌈ 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 *ECUC_CanIf_00772*). ⌋

8.3.22 CanIf_CheckBaudrate

Please note that this API is deprecated and is kept only for backward compatibility reasons. In the next major release this API will be deleted.

[SWS_CANIF_00775] ⌈

Service name:	CanIf_CheckBaudrate	
Syntax:	Std_ReturnType CanIf_CheckBaudrate(uint8 ControllerId, const uint16 Baudrate)	
Service ID[hex]:	0x1c	
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 (in-out):	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 Please note that this API is deprecated and is kept only for backward compatibility reasons. In the next major release this API will be deleted.	

Table 8.27: CanIf_CheckBaudrate

⌋

[SWS_CANIF_00786] ⌈ The service `CanIf_CheckBaudrate()` shall call `Can_CheckBaudrate(Controller, Baudrate)` for the requested CAN controller. ⌋

[SWS_CANIF_00778] ⌈ 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.

[SWS_CANIF_00779] ⌈ Caveats of `CanIf_CheckBaudrate()`:

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

⌋

[SWS_CANIF_00780] ⌈ 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 *ECUC_CanIf_00785*). ⌋

8.3.23 CanIf_ChangeBaudrate

Please note that this API is deprecated and is kept only for backward compatibility reasons. `CanIf_SetBaudrate` API shall be used instead to change the baud rate configuration. In the next major release this API will be deleted.

[SWS_CANIF_00776] ⌈

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 (in-out):	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. Please note that this API is deprecated and is kept only for backward compatibility reasons. CanIf_SetBaudrate API shall be used instead to change the baud rate configuration. In the next major release this API will be deleted.

Table 8.28: CanIf_ChangeBaudrate

⌋

[SWS_CANIF_00787] ⌈ The service `CanIf_ChangeBaudrate()` shall call `Can_ChangeBaudrate(Controller, Baudrate)` for the requested CAN controller. ⌋

[SWS_CANIF_00782] ⌈ 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.

[SWS_CANIF_00783] ⌈ Caveats of `CanIf_ChangeBaudrate()`:

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

⌋

[SWS_CANIF_00784] ⌈ 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 *ECUC_CanIf_00785*). ⌋

8.3.24 CanIf_SetBaudrate

[SWS_CANIF_00867] ⌈

Service name:	CanIf_SetBaudrate	
Syntax:	Std_ReturnType CanIf_SetBaudrate(uint8 ControllerId, uint16 BaudRateConfigID)	
Service ID[hex]:	0x27	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant for different ControllerIds. Non reentrant for the same ControllerId.	
Parameters (in):	ControllerId	CAN controller, whose baud rate shall be set

	BaudRateConfig ID	references a baud rate configuration by ID (see CanControllerBaudRateConfigID)
Parameters (in-out):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Service request accepted, setting of (new) baud rate started E_NOT_OK: Service request not accepted
Description:	This service shall set the baud rate configuration of the CAN controller. Depending on necessary baud rate modifications the controller might have to reset.	

Table 8.29: CanIf_SetBaudrate

⌋

[SWS_CANIF_00868] ⌈ The service `CanIf_SetBaudrate()` shall call `Can_SetBaudrate(Controller, BaudRateConfigID)` for the requested `CAN Controller`. ⌋

[SWS_CANIF_00869] ⌈ If `CanIf_SetBaudrate()` is called with invalid `ControllerId`, `CanIf` shall report development error code `CANIF_E_PARAM_CONTROLLERID` to the `Det_ReportError service` of the DET module. ⌋*(SRS_BSW_00323)*

Note: The parameter `BaudRateConfigID` of `CanIf_SetBaudrate()` is not checked by `CanIf`. This has to be done by responsible `CanDrv`.

[SWS_CANIF_00870] ⌈ Caveats of `CanIf_SetBaudrate()`:

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

⌋

[SWS_CANIF_00871] ⌈ If `CanIf` supports changing baud rate and thus `CanIf_SetBaudrate()`, shall be configurable via `CANIF_SET_BAUDRATE_API` (see *ECUC_CanIf_00838*). ⌋

8.3.25 CanIf_SetIcomConfiguration

[SWS_CANIF_00861] ⌈

Service name:	CanIf_SetIcomConfiguration

Syntax:	Std_ReturnType CanIf_SetIcomConfiguration(uint8 ControllerId, IcomConfigIdType ConfigurationId)	
Service ID[hex]:	0x25	
Sync/Async:	Asynchronous	
Reentrancy:	Reentrant only for different controller Ids	
Parameters (in):	ControllerId	Abstracted CanIf Controller Id which is assigned to a CAN controller.
	ConfigurationId	Requested Configuration
Parameters (in-out):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Request accepted E_NOT_OK: Request denied
Description:	This service shall change the Icom Configuration of a CAN controller to the requested one.	

Table 8.30: CanIf_SetIcomConfiguration

⌋

Note: The interface `CanIf_SetIcomConfiguration()` is called by `CanSm` to activate *Pretended Networking* and load the requested *ICOM* configuration via `CAN Driver`.

[SWS_CANIF_00838] ⌈ The service `CanIf_SetIcomConfiguration()` shall call `Can_SetIcomConfiguration(Controller, ConfigurationId)` for the requested `CanDrv` to set the requested *ICOM configuration*. ⌋

[SWS_CANIF_00872] ⌈ If `CanIf_SetIcomConfiguration()` is called with invalid `ControllerId`, `CanIf` shall report development error code `CANIF_E_PARAM_CONTROLLERID` to the `Det_ReportError service` of the DET module. ⌋*(SRS_BSW_00323)*

[SWS_CANIF_00875] ⌈ `CanIf_SetIcomConfiguration()` shall be pre compile time configurable `ON/OFF` by the configuration parameter `CANIF_PUBLIC_ICOM_SUPPORT` (see *ECUC_CanIf_00839*). ⌋

8.4 Callback notifications

This is a list of functions provided for other modules.

8.4.1 CanIf_TxConfirmation

[SWS_CANIF_00007] ⌈

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 (in-out):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service confirms a previously successfully processed transmission of a CAN TxPDU.	

Table 8.31: CanIf_TxConfirmation

⌋(*SRS_CAN_01009*)

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

Note: Due to the fact `CanDrv` does not support the HandleId concept as described in [15, Specification of ECU Configuration]: Within the service `CanIf_TxConfirmation()`, `CanDrv` uses `PduInfo->swPduHandle` as `CanTxPduId`, which was preserved from `Can_Write(Hth, *PduInfo)`.

[SWS_CANIF_00391] ⌈ If configuration parameters `CANIF_PUBLIC_READTXPDU_NOTIFY_STATUS_API` (*ECUC_CanIf_00609*) and `CANIF_TXPDU_READ_NOTIFYSTATUS` (*ECUC_CanIf_00589*) for the `Transmitted L-PDU` are set to `TRUE`, and if `CanIf_TxConfirmation()` is called, `CanIf` shall set the notification status for the `Transmitted L-PDU`. ⌋

[SWS_CANIF_00410] ⌈ If parameter `CanTxPduId` of `CanIf_TxConfirmation()` has an invalid value, `CanIf` shall report development error code `CANIF_E_PARAM_LPDU` to the `Det_ReportError` service of the DET module, when `CanIf_TxConfirmation()` is called. ⌋(*SRS_BSW_00323*)

[SWS_CANIF_00412] ⌈ If `CanIf` was not initialized before calling `CanIf_TxConfirmation()`, `CanIf` shall not call the service

<User_TxConfirmation>() and shall not set the Tx confirmation status, when CanIf_TxConfirmation() is called. ⌋

[SWS_CANIF_00413] ⌈ 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*.

⌋

[SWS_CANIF_00414] ⌈ Configuration of CanIf_TxConfirmation(): Each Tx L-PDU (see *ECUC_CanIf_00248*) has to be configured with a corresponding transmit confirmation service of an upper layer module (see [SWS_CANIF_00011]) which is called in CanIf_TxConfirmation(). ⌋

8.4.2 CanIf_RxIndication

[SWS_CANIF_00006] ⌈

Service name:	CanIf_RxIndication	
Syntax:	void CanIf_RxIndication(const Can_HwType* Mailbox, const PduInfoType* PduInfoPtr)	
Service ID[hex]:	0x14	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	Mailbox	Identifies the HRH and its corresponding CAN Controller
	PduInfoPtr	Pointer to the received L-PDU
Parameters (in-out):	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.	

Table 8.32: CanIf_RxIndication

⌋

Note: The service CanIf_RxIndication() is implemented in CanIf and called by CanDrv after a CAN L-PDU has been received.

[SWS_CANIF_00415] ⌈ Within the service `CanIf_RxIndication()` the `CanIf` routes this indication to the configured upper layer target service(s). ⌋

[SWS_CANIF_00392] ⌈ If configuration parameters `CANIF_PUBLIC_READRXPDU_NOTIFY_STATUS_API` (*ECUC_CanIf_00608*) and `CANIF_RXPDU_READ_NOTIFYSTATUS` (*ECUC_CanIf_00595*) 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`. ⌋

[SWS_CANIF_00416] ⌈ If parameter `Mailbox->Hoh` of `CanIf_RxIndication()` has an invalid value, `CanIf` shall report development error code `CANIF_E_PARAM_HOH` to the `Det_ReportError` service of the DET module, when `CanIf_RxIndication()` is called. ⌋*(SRS_BSW_00323)*

[SWS_CANIF_00417] ⌈ If parameter `Mailbox->CanId` of `CanIf_RxIndication()` has an invalid value, `CanIf` shall report development error code `CANIF_E_PARAM_CANID` to the `Det_ReportError` service of the DET module, when `CanIf_RxIndication()` is called. ⌋*(SRS_BSW_00323)*

Note: If `CanIf_RxIndication()` is called with invalid `PduInfoPtr->SduLength`, development error `CANIF_E_INVALID_DLC` is reported (see [SWS_CANIF_00168]).

[SWS_CANIF_00419] ⌈ If parameter `PduInfoPtr` or `Mailbox` of `CanIf_RxIndication()` has an invalid value, `CanIf` shall report development error code `CANIF_E_PARAM_POINTER` to the `Det_ReportError` service of the DET module, when `CanIf_RxIndication()` is called. ⌋*(SRS_BSW_00323)*

[SWS_CANIF_00421] ⌈ If `CanIf` was not initialized before calling `CanIf_RxIndication()`, `CanIf` shall not execute *Rx indication handling*, when `CanIf_RxIndication()`, is called. ⌋

[SWS_CANIF_00422] ⌈ 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*.

⌋

[SWS_CANIF_00423] ⌈ Configuration of `CanIf_RxIndication()`: Each `Rx L-PDU` (see *ECUC_CanIf_00249*) has to be configured with a corresponding receive indication service of an upper layer module (see [SWS_CANIF_00012]) which is called in `CanIf_RxIndication()`. ⌋

8.4.3 CanIf_CancelTxConfirmation

[SWS_CANIF_00101] ⌈

Service name:	CanIf_CancelTxConfirmation

Syntax:	void CanIf_CancelTxConfirmation(const Can_PduType* CanPduPtr)	
Service ID[hex]:	0x15	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	CanPduPtr	Pointer to a structure with CAN L-PDU related data: L-PDU handle of the successfully aborted CAN L-SDU, CAN identifier, DLC and pointer to CAN L-SDU buffer.
Parameters (in-out):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service informs CanIf that a L-PDU shall be buffered in CanIf TxBuffer from CAN hardware object to avoid priority inversion.	

Table 8.33: CanIf_CancelTxConfirmation

⌋

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

[SWS_CANIF_00828] ⌈ If `CanIf_CancelTxConfirmation()` is called with invalid `CanPduPtr`, `CanIf` shall report development error code `CANIF_E_PARAM_POINTER` to the `Det_ReportError` service of the DET module. ⌋*(SRS_BSW_00323)*

[SWS_CANIF_00424] ⌈ If `CanIf_CancelTxConfirmation()` is called with invalid `CanPduPtr->id`, `CanIf` shall report development error code `CANIF_E_PARAM_LPDU` to the `Det_ReportError` service of the DET module. ⌋*(SRS_BSW_00323)*

[SWS_CANIF_00426] ⌈ If `CanIf` was not initialized, `CanIf` shall not execute *Tx Cancellation handling*, when `CanIf_CancelTxConfirmation()` is called. ⌋

[SWS_CANIF_00427] ⌈ Caveats of `CanIf_CancelTxConfirmation()`:

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

⌋

[SWS_CANIF_00428] ⌈ Configuration of `CanIf_CancelTxConfirmation()`: This function shall be pre compile time configurable `ON/OFF` by the configuration parameter `CANIF_CTRLDRV_TX_CANCELLATION` (see *ECUC_CanIf_00640*). ⌋

8.4.4 CanIf_ControllerBusOff

[SWS_CANIF_00218] ⌈

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 (in-out):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service indicates a Controller BusOff event referring to the corresponding CAN Controller.	

Table 8.34: CanIf_ControllerBusOff

⌋

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

[SWS_CANIF_00429] ⌈ If parameter `ControllerId` of `CanIf_ControllerBusOff()` has an invalid value, `CanIf` shall report development error code `CANIF_E_PARAM_CONTROLLER` to the `Det_ReportError` service of the DET module, when `CanIf_ControllerBusOff()` is called. ⌋*(SRS_BSW_00323)*

[SWS_CANIF_00431] ⌈ If `CanIf` was not initialized before calling `CanIf_ControllerBusOff()`, `CanIf` shall not execute *BusOff notification*, when `CanIf_ControllerBusOff()`, is called. ⌋

[SWS_CANIF_00432] ⌈ 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*.

⌋

[SWS_CANIF_00433] ⌈ Configuration of `CanIf_ControllerBusOff()`: `ID` of the `CAN Controller` is published inside the configuration description of the `CanIf` (see *ECUC_CanIf_00546*). ⌋

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

8.4.5 CanIf_ConfirmPnAvailability

[SWS_CANIF_00815] ⌈

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 (in-out):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service indicates that the transceiver is running in PN communication mode.	

Table 8.35: CanIf_ConfirmPnAvailability

⌋

[SWS_CANIF_00753] ⌈ If `CanIf_ConfirmPnAvailability()` is called, `CanIf` calls `<User_ConfirmPnAvailability>()`. ⌋

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

[SWS_CANIF_00816] ⌈ If parameter `TransceiverId` of `CanIf_ConfirmPnAvailability()` has an invalid value, `CanIf` shall report development error code `CANIF_E_PARAM_TRCV` to the `Det_ReportError` service of the DET module, when `CanIf_ConfirmPnAvailability()` is called. ⌋

[SWS_CANIF_00817] ⌈ If CanIf was not initialized before calling CanIf_ConfirmPnAvailability(), CanIf shall not execute notification, when CanIf_ConfirmPnAvailability() is called. ⌋

[SWS_CANIF_00818] ⌈ 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*.

⌋

[SWS_CANIF_00754] ⌈ Configuration of CanIf_ConfirmPnAvailability(): This function shall be pre compile time configurable ON/OFF by the configuration parameter CANIF_PUBLIC_PN_SUPPORT (see *ECUC_CanIf_00772*). ⌋

8.4.6 CanIf_ClearTrcvWufFlagIndication

[SWS_CANIF_00762] ⌈

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 (in-out):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service indicates that the transceiver has cleared the WufFlag.	

Table 8.36: CanIf_ClearTrcvWufFlagIndication

⌋

[SWS_CANIF_00757] ⌈ If CanIf_ClearTrcvWufFlagIndication() is called, CanIf calls <User_ClearTrcvWufFlagIndication>(). ⌋

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

[SWS_CANIF_00805] ⌈ If parameter `TransceiverId` of `CanIf_ClearTrcvWufFlagIndication()` has an invalid value, `CanIf` shall report development error code `CANIF_E_PARAM_TRCV` to the `Det_ReportError` service of the DET module, when `CanIf_ClearTrcvWufFlagIndication()` is called. ⌋

[SWS_CANIF_00806] ⌈ If `CanIf` was not initialized before calling `CanIf_ClearTrcvWufFlagIndication()`, `CanIf` shall not execute notification, when `CanIf_ClearTrcvWufFlagIndication()` is called. ⌋

[SWS_CANIF_00807] ⌈ 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*.

⌋

[SWS_CANIF_00808] ⌈ Configuration of `CanIf_ClearTrcvWufFlagIndication()`: This function shall be pre compile time configurable `ON/OFF` by the configuration parameter `CANIF_PUBLIC_PN_SUPPORT` (see *ECUC_CanIf_00772*). ⌋

8.4.7 CanIf_CheckTrcvWakeFlagIndication

[SWS_CANIF_00763] ⌈

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 (in-out):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service indicates the reason for the wake up that the CAN transceiver has detected.	

Table 8.37: CanIf_CheckTrcvWakeFlagIndication

⌋

[SWS_CANIF_00759] ⌈ If `CanIf_CheckTrcvWakeFlagIndication()` is called, `CanIf` calls `<User_CheckTrcvWakeFlagIndication>()`. ⌋

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

[SWS_CANIF_00809] ⌈ If parameter `TransceiverId` of `CanIf_CheckTrcvWakeFlagIndication()` has an invalid value, `CanIf` shall report development error code `CANIF_E_PARAM_TRCV` to the `Det_ReportError` service of the DET module, when `CanIf_CheckTrcvWakeFlagIndication()` is called. ⌋

[SWS_CANIF_00810] ⌈ If the CanIf was not initialized before calling `CanIf_CheckTrcvWakeFlagIndication()`, `CanIf` shall not execute notification, when `CanIf_CheckTrcvWakeFlagIndication()` is called. ⌋

[SWS_CANIF_00811] ⌈ 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*.

⌋

[SWS_CANIF_00812] ⌈ Configuration of `CanIf_CheckTrcvWakeFlagIndication()`: This function shall be pre compile time configurable `ON/OFF` by the configuration parameter `CANIF_PUBLIC_PN_SUPPORT` (see *ECUC_CanIf_00772*). ⌋

8.4.8 CanIf_ControllerModeIndication

[SWS_CANIF_00699] ⌈

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 (in-out):	None	
Parameters (out):	None	

Return value:	None
Description:	This service indicates a controller state transition referring to the corresponding CAN controller.

Table 8.38: CanIf_ControllerModeIndication

⌋

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

[SWS_CANIF_00700] ⌈ If parameter `ControllerId` of `CanIf_ControllerModeIndication()` has an invalid value, `CanIf` shall report development error code `CANIF_E_PARAM_CONTROLLER` to the `Det_ReportError` service of the DET module, when `CanIf_ControllerModeIndication()` is called. ⌋

[SWS_CANIF_00702] ⌈ If `CanIf` was not initialized before calling `CanIf_ControllerModeIndication()`, `CanIf` shall not execute state transition notification, when `CanIf_ControllerModeIndication()` is called. ⌋

[SWS_CANIF_00703] ⌈ 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*.

⌋

[SWS_CANIF_00704] ⌈ Configuration of `CanIf_ControllerModeIndication()`: `ID` of the `CAN Controller` is published inside the configuration description of `CanIf` (see *ECUC_CanIf_00647*). ⌋

8.4.9 CanIf_TrcvModeIndication

[SWS_CANIF_00764] ⌈

Service name:	CanIf_TrcvModeIndication
Syntax:	void CanIf_TrcvModeIndication(uint8 TransceiverId, CanTrcv_TrcvModeType TransceiverMode)
Service ID[hex]:	0x22
Sync/Async:	Synchronous
Reentrancy:	Reentrant

Parameters (in):	TransceiverId	CAN transceiver, which state has been transitioned.
	TransceiverMode	Mode to which the CAN transceiver transitioned
Parameters (in-out):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service indicates a transceiver state transition referring to the corresponding CAN transceiver.	

Table 8.39: CanIf_TrcvModeIndication

⌋

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

[SWS_CANIF_00706] ⌈ If parameter `TransceiverId` of `CanIf_TrcvModeIndication()` has an invalid value, `CanIf` shall report development error code `CANIF_E_PARAM_TRCV` to the `Det_ReportError` service of the DET module, when `CanIf_TrcvModeIndication()` is called. ⌋

[SWS_CANIF_00708] ⌈ If `CanIf` was not initialized before calling `CanIf_TrcvModeIndication()`, `CanIf` shall not execute state transition notification, when `CanIf_TrcvModeIndication()` is called. ⌋

[SWS_CANIF_00709] ⌈ 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*.

⌋

[SWS_CANIF_00710] ⌈ Configuration of `CanIf_TrcvModeIndication()`: `ID` of the `CAN Transceiver` is published inside the configuration description of `CanIf` via parameter `CANIF_TRCV_ID` (see *ECUC_CanIf_00654*). ⌋

[SWS_CANIF_00730] ⌈ Configuration of `CanIf_TrcvModeIndication()`: If transceivers are not supported (`CanIfTrcvDrvCfg` is not configured, see *ECUC_CanIf_00273*), `CanIf_TrcvModeIndication()` shall not be provided by `CanIf`. ⌋

8.4.10 CanIf_CurrentIcomConfiguration

[SWS_CANIF_00862] ⌈

Service name:	CanIf_CurrentIcomConfiguration	
Syntax:	void CanIf_CurrentIcomConfiguration(uint8 ControllerId, IcomConfigIdType ConfigurationId, IcomSwitch_ErrorType Error)	
Service ID[hex]:	0x26	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant only for different controller Ids	
Parameters (in):	ControllerId	CAN controller Id, which informs about the Configuration Id.
	ConfigurationId Error	Active Configuration Id. ICOM_SWITCH_E_OK: No Error ICOM_SWITCH_E_FAILED: Switch to requested Configuration failed. Severe Error.
Parameters (in-out):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service shall inform about the change of the Icom Configuration of a CAN controller.	

Table 8.40: CanIf_CurrentIcomConfiguration

⌋

Note: The interface `CanIf_CurrentIcomConfiguration()` is used by the `CanDrv` to inform `CanIf` about the status of activation or deactivation of *Pretended Networking* for a given channel.

[SWS_CANIF_00839] ⌈ If `CanIf_CurrentIcomConfiguration()` is called, `CanIf` shall call `CanSM_CurrentIcomConfiguration(ControllerId, ConfigurationId, Error)` to inform `CanSM` about current status of *ICOM*. ⌋

[SWS_CANIF_00873] ⌈ If `CanIf_CurrentIcomConfiguration()` is called with invalid `ControllerId`, `CanIf` shall report development error code `CANIF_E_PARAM_CONTROLLERID` to the `Det_ReportError service` of the DET module. ⌋*(SRS_BSW_00323)*

[SWS_CANIF_00876] ⌈ `CanIf_CurrentIcomConfiguration()` shall be pre compile time configurable `ON/OFF` by the configuration parameter `CANIF_PUBLIC_ICOM_SUPPORT` (see *ECUC_CanIf_00839*). ⌋

8.5 Scheduled functions

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

[SWS_CANIF_00040] ⌈

API function	*Description*
Can_SetControllerMode	This function performs software triggered state transitions of the CAN controller State machine.
Can_Write	This function is called by CanIf to pass a CAN message to CanDrv for transmission.
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 section defines all interfaces, which are required to fulfill an optional functionality of the module.

[SWS_CANIF_00294] ⌈

API function	*Description*
CanSM_CurrentIcomConfiguration	This service shall inform about the change of the Icom Configuration of a CAN network.
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. Please note that this API is deprecated and is kept only for backward compatibility reasons. Can_SetBaudrate API shall be used instead to change the baud rate configuration. In the next major release this API will be deleted.
Can_CheckBaudrate	This service shall check, if a certain CAN controller supports a requested baudrate Please note that this API is deprecated and is kept only for backward compatibility reasons. In the next major release this API will be deleted.
Can_CheckWakeup	This function checks if a wakeup has occurred for the given controller.
Can_SetBaudrate	This service shall set the baud rate configuration of the CAN controller. Depending on necessary baud rate modifications the controller might have to reset.
Can_SetIcomConfiguration	This service shall change the Icom Configuration of a CAN controller to the requested one.
Det_ReportError	Service to report development errors.

⌋

8.6.3 Configurable interfaces

In this section 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 `CanIf` 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 section describes only the content of notification of the callback, the call context inside `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 `CanIf`. 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-SDU ID.

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

8.6.3.1 <User_TxConfirmation>

[SWS_CANIF_00011] ⌈

Service name:	<User_TxConfirmation>	
Syntax:	void <User_TxConfirmation>(PduIdType TxPduId)	
Service ID[hex]:	–	
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 (in-out):	None	
Parameters (out):	None	
Return value:	None	
Description:	The lower layer communication interface module confirms the transmission of an I-PDU.	

Table 8.43: <User_TxConfirmation>

⌋

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

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

Note: Parameter TxPduId is derived from <User> configuration.

[SWS_CANIF_00437] ⌈ Caveats of <User_TxConfirmation>(): The call context is either on interrupt level (*interrupt mode*) or on task level (*polling mode*). ⌋

[SWS_CANIF_00438] ⌈ Configuration of <User_TxConfirmation>(): The upper layer module, which provides this callback service, has to be configured by CANIF_TXPDU_USERTXCONFIRMATION_UL (see *ECUC_CanIf_00527*). If no upper layer modules are configured for *transmit confirmation* using <User_TxConfirmation>(), no *transmit confirmation* is executed. ⌋

[SWS_CANIF_00542] ⌈ Configuration of `<User_TxConfirmation>()`: The name of the API `<User_TxConfirmation>()` which is called by `CanIf` shall be configured for `CanIf` by parameter `CANIF_TXPDU_USERTXCONFIRMATION_NAME` (see *ECUC_CanIf_00528*). ⌋

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.

[SWS_CANIF_00439] ⌈ Configuration of `<User_TxConfirmation>()`: If `CANIF_TXPDU_USERTXCONFIRMATION_UL` is set to `PDUR`, `CANIF_TXPDU_USERTXCONFIRMATION_NAME` must be `PduR_CanIfTxConfirmation`. ⌋

[SWS_CANIF_00543] ⌈ Configuration of `<User_TxConfirmation>()`: If `CANIF_TXPDU_USERTXCONFIRMATION_UL` is set to `CAN_NM`, `CANIF_TXPDU_USERTXCONFIRMATION_NAME` must be `CanNm_TxConfirmation`. ⌋

Hint (Dependency to another module):
If at least one `CanIf` Tx `L-SDU` 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` (for `CanNm` related details see [4, Specification of CAN Network Management], [SWS_CANNM_00284]).

[SWS_CANIF_00858] ⌈ Configuration of `<User_TxConfirmation>()`: If `CANIF_TXPDU_USERTXCONFIRMATION_UL` is set to `J1939NM`, `CANIF_TXPDU_USERTXCONFIRMATION_NAME` must be `J1939Nm_TxConfirmation`. ⌋

[SWS_CANIF_00544] ⌈ Configuration of `<User_TxConfirmation>()`: If `CANIF_TXPDU_USERTXCONFIRMATION_UL` is set to `J1939TP`, `CANIF_TXPDU_USERTXCONFIRMATION_NAME` must be `J1939Tp_TxConfirmation`. ⌋

[SWS_CANIF_00550] ⌈ Configuration of `<User_TxConfirmation>()`: If `CANIF_TXPDU_USERTXCONFIRMATION_UL` is set to `CAN_TP`, `CANIF_TXPDU_USERTXCONFIRMATION_NAME` must be `CanTp_TxConfirmation`. ⌋

[SWS_CANIF_00556] ⌈ Configuration of `<User_TxConfirmation>()`: If `CANIF_TXPDU_USERTXCONFIRMATION_UL` is set to `XCP`, `CANIF_TXPDU_USERTXCONFIRMATION_NAME` must be `Xcp_CanIfTxConfirmation`. ⌋

[SWS_CANIF_00551] ⌈ 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>

[SWS_CANIF_00012] ⌈

Service name:	<User_RxIndication>	
Syntax:	void <User_RxIndication>(PduIdType RxPduId, const PduInfoType* PduInfoPtr)	
Service ID[hex]:	–	
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 (in-out):	None	
Parameters (out):	None	
Return value:	None	
Description:	Indication of a received I-PDU from a lower layer communication interface module.	

Table 8.44: <User_RxIndication>

⌋*(SRS_CAN_01003)*

Note: This service indicates a successful *reception* of an *L-SDU* to the upper layer module after passing all filters and validation checks.

Note: This callback service is called by CanIf and implemented in the configured upper layer module (e.g. PduR, CanNm, CanTp, etc.) if configured accordingly (see *ECUC_CanIf_00529*).

Note: Besides the L-SDU the buffer referenced by parameter PduInfoPtr->SduDataPtr also contains the MetaData of dynamic L-SDUs.

[SWS_CANIF_00440] ⌈ Caveats of <User_RxIndication>:

- Until this service returns, CanIf will not access <PduInfoPtr>. The <PduInfoPtr> is only valid and can be used by upper layers, until the indication returns.

CanIf guarantees that the number of configured bytes for this <PduInfoPtr> is valid.

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

⌋

[SWS_CANIF_00441] ⌈ Configuration of <User_RxIndication>(): The upper layer module, which provides this callback service, has to be configured by CANIF_RXPDU_USERRXINDICATION_UL (see *ECUC_CanIf_00529*). ⌋

[SWS_CANIF_00552] ⌈ Configuration of <User_RxIndication>(): The name of the API <User_RxIndication>() which will be called by CanIf shall be configured for CanIf by parameter CANIF_RXPDU_USERRXINDICATION_NAME (see *ECUC_CanIf_00530*). ⌋

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.

[SWS_CANIF_00442] ⌈ Configuration of <User_RxIndication>(): If CANIF_RXPDU_USERRXINDICATION_UL is set to PDUR, CANIF_RXPDU_USERRXINDICATION_NAME must be PduR_CanIfRxIndication. ⌋

[SWS_CANIF_00445] ⌈ Configuration of <User_RxIndication>(): If CANIF_RXPDU_USERRXINDICATION_UL is set to CAN_NM, CANIF_RXPDU_USERRXINDICATION_NAME must be CanNm_RxIndication. ⌋

The value passed to CanNm via the API parameter CanNmRxPduId refers to the CanNm channel handle within the CanNm module (for CanNm related details see [4, Specification of CAN Network Management]).

[SWS_CANIF_00859] ⌈ Configuration of <User_RxIndication>(): If CANIF_RXPDU_USERRXINDICATION_UL is set to J1939NM, CANIF_RXPDU_USERRXINDICATION_NAME must be J1939Nm_RxIndication. ⌋

[SWS_CANIF_00448] ⌈ Configuration of <User_RxIndication>(): If CANIF_RXPDU_USERRXINDICATION_UL is set to CAN_TP, CANIF_RXPDU_USERRXINDICATION_NAME must be CanTp_RxIndication. ⌋

[SWS_CANIF_00554] ⌈ Configuration of <User_RxIndication>(): If CANIF_RXPDU_USERRXINDICATION_UL is set to J1939TP, CANIF_RXPDU_USERRXINDICATION_NAME must be J1939Tp_RxIndication. ⌋

[SWS_CANIF_00555] ⌈ Configuration of `<User_RxIndication>()`: If `CANIF_RXPDU_USERRXINDICATION_UL` is set to `XCP`, `CANIF_RXPDU_USERRXINDICATION_NAME` must be `Xcp_CanIfRxIndication`. ⌋

[SWS_CANIF_00557] ⌈ 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>

[SWS_CANIF_00532] ⌈

Service name:	<User_ValidateWakeupEvent>	
Syntax:	void <User_ValidateWakeupEvent>(EcuM_WakeupSourceType sources)	
Service ID[hex]:	–	
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 (in-out):	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.	

Table 8.45: User_ValidateWakeupEvent

⌋

Note: This callback service is mainly implemented in and used by the *ECU State Manager* module (see [14, Specification of ECU State Manager]).

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`.

[SWS_CANIF_00455] ⌈ 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.

⌋

[SWS_CANIF_00659] ⌈ Configuration of <User_ValidateWakeupEvent>(): If no validation is needed, this API can be omitted by disabling CANIF_PUBLIC_WAKEUP_CHECK_VALIDATION_SUPPORT (see *ECUC_CanIf_00611*). ⌋

[SWS_CANIF_00456] ⌈ Configuration of <User_ValidateWakeupEvent>(): The upper layer module which provides this callback service has to be configured by CANIF_DISPATCH_USERVALIDATEWAKEUPEVENT_UL (see *ECUC_CanIf_00549*), 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 *ECUC_CanIf_00637*, *ECUC_CanIf_00606*), CANIF_DISPATCH_USERVALIDATEWAKEUPEVENT_UL is not configurable.

⌋

[SWS_CANIF_00563] ⌈ Configuration of <User_ValidateWakeupEvent>(): If CANIF_DISPATCH_USERVALIDATEWAKEUPEVENT_UL is set to ECUM, CANIF_DISPATCH_USERVALIDATEWAKEUPEVENT_NAME must be EcuM_ValidateWakeupEvent. ⌋

[SWS_CANIF_00564] ⌈ 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>

[SWS_CANIF_00014] ⌈

Service name:	<User_ControllerBusOff>
Syntax:	void <User_ControllerBusOff>(uint8 ControllerId)

Service ID[hex]:	–	
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 (in-out):	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).	

Table 8.46: User_ControllerBusOff

⌋(*SRS_CAN_01029*)

Note: This callback service is mainly implemented in and used by CanSm (see [3, Specification of CAN State Manager]).

Note: This callback service is called by 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.

[SWS_CANIF_00449] ⌈ 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.

⌋

Configuration of <User_ControllerBusOff>()

[SWS_CANIF_00450] ⌈ Configuration of <User_ControllerBusOff>():
The upper layer module which provides this callback service has to be configured by CANIF_DISPATCH_USERCTRLBUSOFF_UL (see *ECUC_CanIf_00547*). ⌋

[SWS_CANIF_00558] ⌈ Configuration of `<User_ControllerBusOff>()`: The name of the API `<User_ControllerBusOff>()` which will be called by `CanIf` shall be configured for `CanIf` by parameter `CANIF_DISPATCH_USERCTRLBUSOFF_NAME` (see *ECUC_CanIf_00525*). ⌋

[SWS_CANIF_00524] ⌈ Configuration of `<User_ControllerBusOff>()`: At least one upper layer module and hence an API of `<User_ControllerBusOff>()` has mandatorily to be configured, which `CanIf` can call in case of an occurred call of `CanIf_ControllerBusOff()`. ⌋

[SWS_CANIF_00559] ⌈ Configuration of `<User_ControllerBusOff>()`: If `CANIF_DISPATCH_USERCTRLBUSOFF_UL` is set to `CAN_SM`, `CANIF_DISPATCH_USERCTRLBUSOFF_NAME` must be `CanSM_ControllerBusOff`. ⌋

[SWS_CANIF_00560] ⌈ 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>

[SWS_CANIF_00821] ⌈

Service name:	<User_ConfirmPnAvailability>	
Syntax:	void <User_ConfirmPnAvailability>(uint8 TransceiverId)	
Service ID[hex]:	–	
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 (in-out):	None	
Parameters (out):	None	
Return value:	None	
Description:	This service indicates that the CAN transceiver is running in PN communication mode.	

Table 8.47: User_ConfirmPnAvailability

⌋

Note: This callback service is mainly implemented in and used by CanSm (see [3, Specification of CAN State Manager]).

[SWS_CANIF_00822] ⌈ Caveats of <User_ConfirmPnAvailability>():

- The CanTrcv 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.

⌋

[SWS_CANIF_00823] ⌈ Configuration of <User_ConfirmPnAvailability>(): The upper layer module, which is called (see [SWS_CANIF_00753]), has to be configurable by CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_UL (see *ECUC_CanIf_00820*) if CANIF_PUBLIC_PN_SUPPORT (see *ECUC_CanIf_00772*) equals True. ⌋

[SWS_CANIF_00824] ⌈ Configuration of <User_ConfirmPnAvailability>(): The name of <User_ConfirmPnAvailability>() shall be configurable by CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_NAME (see *ECUC_CanIf_00819*) if CANIF_PUBLIC_PN_SUPPORT (see *ECUC_CanIf_00772*) equals True. ⌋

[SWS_CANIF_00825] ⌈ Configuration of <User_ConfirmPnAvailability>(): It shall be configurable by CANIF_PUBLIC_PN_SUPPORT (see *ECUC_CanIf_00772*), if CanIf supports this service (False: not supported, True: supported) ⌋

[SWS_CANIF_00826] ⌈ Configuration of <User_ConfirmPnAvailability>(): If CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_UL is set to CAN_SM, CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_NAME must be CanSM_ConfirmPnAvailability. ⌋

[SWS_CANIF_00827] ⌈ Configuration of <User_ConfirmPnAvailability>(): If CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_UL is set to CDD, the name of the service has to be configurable via parameter CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_NAME and the function parameter has to be of type uint8. ⌋

8.6.3.6 <User_ClearTrcvWufFlagIndication>

[SWS_CANIF_00788] ⌈

Service name:	<User_ClearTrcvWufFlagIndication>
Syntax:	void <User_ClearTrcvWufFlagIndication>(uint8 TransceiverId)

Service ID[hex]:	–	
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.	

Table 8.48: <User_ClearTrcvWufFlagIndication>

⌋

Note: This callback service is mainly implemented in and used by CanSm (see [3, Specification of CAN State Manager]).

[SWS_CANIF_00793] ⌈ Caveats of <User_ClearTrcvWufFlagIndication>():

- The CanTrcv 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.

⌋

[SWS_CANIF_00794] ⌈ Configuration of <User_ClearTrcvWufFlagIndication>(): The upper layer module, which is called (see [SWS_CANIF_00757]), has to be configurable by CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_UL (see *ECUC_CanIf_00790*) if CANIF_PUBLIC_PN_SUPPORT (see *ECUC_CanIf_00772*) equals True. ⌋

[SWS_CANIF_00795] ⌈ Configuration of <User_ClearTrcvWufFlagIndication>(): The name of <User_ClearTrcvWufFlagIndication>() shall be configurable by CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_NAME (see *ECUC_CanIf_00789*) if CANIF_PUBLIC_PN_SUPPORT (see *ECUC_CanIf_00772*) equals True. ⌋

[SWS_CANIF_00796] ⌈ Configuration of <User_ClearTrcvWufFlagIndication>(): It shall be configurable by

CANIF_PUBLIC_PN_SUPPORT (see *ECUC_CanIf_00772*), if CanIf supports this service (False: not supported, True: supported) ⌋

[SWS_CANIF_00797] ⌈ Configuration of <User_ClearTrcvWufFlagIndication>(): If CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_UL is set to CAN_SM, CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_NAME must be CanSM_ClearTrcvWufFlagIndication. ⌋

[SWS_CANIF_00798] ⌈ Configuration of <User_ClearTrcvWufFlagIndication>(): If CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_UL is set to CDD, the name of the service has to be configurable via parameter CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_NAME and the function parameter has to be of type uint8. ⌋

8.6.3.7 <User_CheckTrcvWakeFlagIndication>

[SWS_CANIF_00814] ⌈

Service name:	<User_CheckTrcvWakeFlagIndication>	
Syntax:	void <User_CheckTrcvWakeFlagIndication>(uint8 TransceiverId)	
Service ID[hex]:	–	
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.	

Table 8.49: <User_CheckTrcvWakeFlagIndication>

⌋

Note: This callback service is mainly implemented in and used by CanSm (see [3, Specification of CAN State Manager]).

[SWS_CANIF_00799] ⌈ Caveats of <User_CheckTrcvWakeFlagIndication>():

- The CanTrcv 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.

⌋

[SWS_CANIF_00800] ⌈ Configuration of <User_CheckTrcvWakeFlagIndication>(): The upper layer module, which is called (see [SWS_CANIF_00759]), has to be configurable by CANIF_DISPATCH_USERCHECKRCVWAKEFLAGINDICATION_UL (see *ECUC_CanIf_00792*) if CANIF_PUBLIC_PN_SUPPORT (see *ECUC_CanIf_00772*) equals True. ⌋

[SWS_CANIF_00801] ⌈ Configuration of <User_CheckTrcvWakeFlagIndication>(): The name of <User_CheckTrcvWakeFlagIndication>() shall be configurable by CANIF_DISPATCH_USERCHECKRCVWAKEFLAGINDICATION_NAME (see *ECUC_CanIf_00791*) if CANIF_PUBLIC_PN_SUPPORT (see *ECUC_CanIf_00772*) equals True. ⌋

[SWS_CANIF_00802] ⌈ Configuration of <User_CheckTrcvWakeFlagIndication>(): It shall be configurable by CANIF_PUBLIC_PN_SUPPORT (see *ECUC_CanIf_00772*), if CanIf supports this service (False: not supported, True: supported) ⌋

[SWS_CANIF_00803] ⌈ Configuration of <User_CheckTrcvWakeFlagIndication>():
If CANIF_DISPATCH_USERCHECKRCVWAKEFLAGINDICATION_UL is set to CAN_SM, CANIF_DISPATCH_USERCHECKRCVWAKEFLAGINDICATION_NAME must be CanSM_CheckTrcvWakeFlagIndication. ⌋

[SWS_CANIF_00804] ⌈ Configuration of <User_CheckTrcvWakeFlagIndication>():
If CANIF_DISPATCH_USERCHECKRCVWAKEFLAGINDICATION_UL is set to CDD, the name of the service has to be configurable via parameter CANIF_DISPATCH_USERCHECKRCVWAKEFLAGINDICATION_NAME and the function parameter has to be of type uint8. ⌋

8.6.3.8 <User_ControllerModeIndication>

[SWS_CANIF_00687] ⌈

Service name:	<User_ControllerModeIndication>

Syntax:	void <User_ControllerModeIndication>(uint8 ControllerId, CanIf_ControllerModeType ControllerMode)	
Service ID[hex]:	–	
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 (in-out):	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).	

Table 8.50: <User_ControllerModeIndication>

⌋

Note: The upper layer module provides the Service ID.

Note: This callback service is mainly implemented in and used by CanSm (see [3, Specification of CAN State Manager]).

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.

[SWS_CANIF_00688] ⌈ 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.

⌋

[SWS_CANIF_00689] ⌈ Configuration of `<User_ControllerModeIndication>()`: The upper layer module which provides this callback service has to be configured by `CANIF_USERCONTROLLERMODEINDICATION_UL` (see *ECUC_CanIf_00684*). ⌋

[SWS_CANIF_00690] ⌈ Configuration of `<User_ControllerModeIndication>()`: The name of `<User_ControllerModeIndication>()` which is called by `CanIf` shall be configured for `CanIf` by parameter `CANIF_DISPATCH_USERCTRLMODEINDICATION_NAME` (see *ECUC_CanIf_00683*). This is only necessary if *state transition notifications* are configured via `CANIF_DISPATCH_USERCTRLMODEINDICATION_UL`. ⌋

[SWS_CANIF_00691] ⌈ Configuration of `<User_ControllerModeIndication>()`:
If `CANIF_DISPATCH_USERCTRLMODEINDICATION_UL` is set to `CAN_SM`, `CANIF_DISPATCH_USERCTRLMODEINDICATION_NAME` must be `CanSM_ControllerModeIndication`. ⌋

[SWS_CANIF_00692] ⌈ 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>

[SWS_CANIF_00693] ⌈

Service name:	<User_TrcvModeIndication>	
Syntax:	void <User_TrcvModeIndication>(uint8 TransceiverId, CanTrcv_TrcvModeType TransceiverMode)	
Service ID[hex]:	–	
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).

Table 8.51: <User_TrcvModeIndication>

⌋

Note: The upper layer module provides the Service ID.

Note: This callback service is mainly implemented in and used by CanSm (see [3, Specification of CAN State Manager]).

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.

[SWS_CANIF_00694] ⌈ 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.

⌋

[SWS_CANIF_00695] ⌈ Configuration of <User_TrcvModeIndication>():
The upper layer module which provides this callback service has to be configured by CANIF_DISPATCH_USERTRCVMODEINDICATION_UL (see *ECUC_CanIf_00686*), 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 is not configured, see *ECUC_CanIf_00273*), 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.

[SWS_CANIF_00696] ⌈ Configuration of `<User_TrcvModeIndication>()`: The name of `<User_TrcvModeIndication>()` which will be called by `CanIf` shall be configured for `CanIf` by parameter `CANIF_DISPATCH_USERTRCVMODEINDICATION_NAME` (see *ECUC_CanIf_00685*). This is only necessary if *state transition notifications* are configured via `CANIF_DISPATCH_USERTRCVMODEINDICATION_UL`. ⌋

[SWS_CANIF_00697] ⌈ Configuration of `<User_TrcvModeIndication>()`: If `CANIF_DISPATCH_USERTRCVMODEINDICATION_UL` is set to `CAN_SM`, `CANIF_DISPATCH_USERTRCVMODEINDICATION_NAME` must be `CanSM_TransceiverModeIndication`. ⌋

[SWS_CANIF_00698] ⌈ 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)

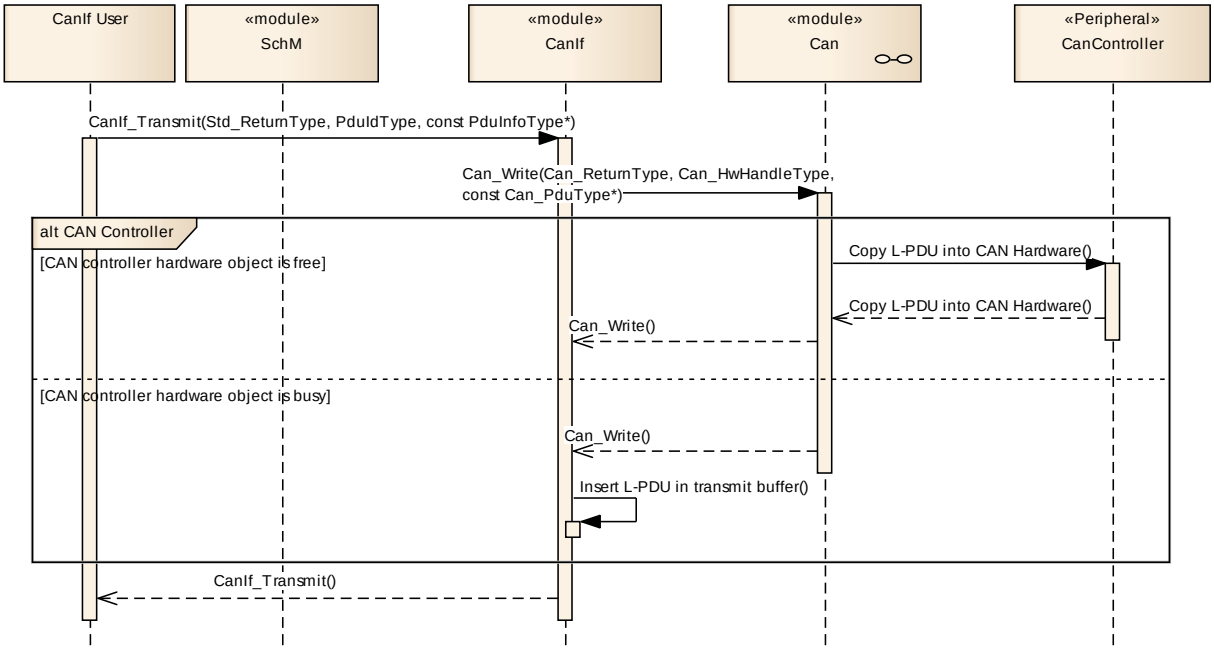

Figure 9.1: Transmission request with a single CAN Driver

Activity	Description
Transmission request	The upper layer initiates a transmit request via the service CanIf_Transmit(). The parameter CanTxPduId identifies the requested L-SDU. 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-SDU related data such as SduLength and *SduDataPtr.
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 CanDrv detects, there are no free hardware objects available, it returns CAN_E_BUSY to 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 CanIf	CanIf_Transmit() returns E_OK to the upper layer.

9.2 Transmit request (multiple CAN Drivers)

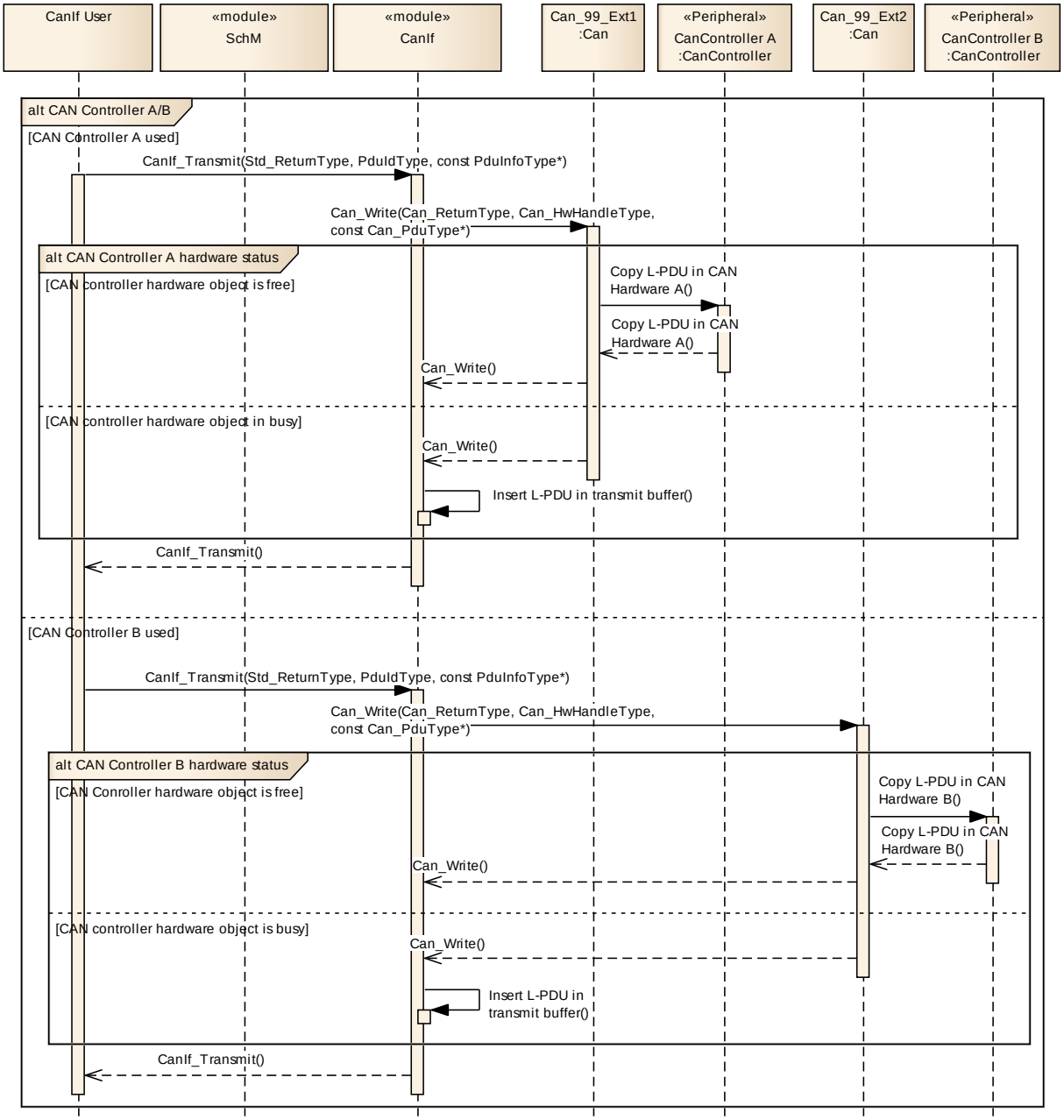

Figure 9.2: Transmission request with multiple CAN Drivers

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-SDU`. 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-SDU` related data such as `SduLength` and `*SduDataPtr`.
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 `CanDrv` `Can_99_Ext1` detects, there are no free hardware objects available, it returns `CAN_E_BUSY` to `CanIf`.
Copying into the buffer	The `L-PDU` of the rejected transmit request will be inserted in the transmit buffers of `CanIf` until the next transmit confirmation.
`E_OK` from `CanIf`	`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-SDU`. 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 transmit `L-SDU` related data such as `SduLength` and `*SduDataPtr`.
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 `CanDrv` `Can_99_Ext2` detects, there are no free hardware objects available, it returns `CAN_E_BUSY` to `CanIf`.
Copying into the buffer	The `L-PDU` of the rejected transmit request will be inserted in the transmit buffers of `CanIf` until the next transmit confirmation.
`E_OK` from `CanIf`	`CanIf_Transmit()` returns `E_OK` to the upper layer.

9.3 Transmit confirmation (interrupt mode)

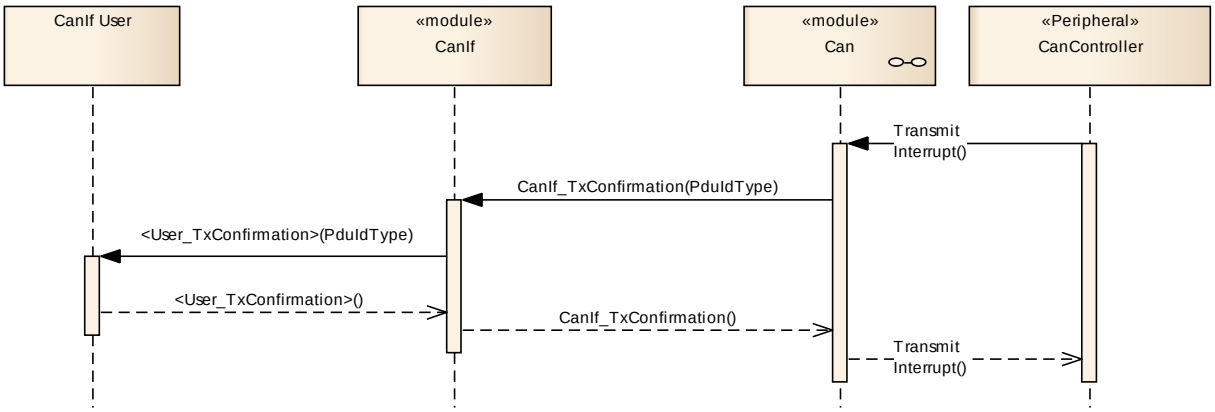

Figure 9.3: Transmit confirmation interrupt driven

Activity	Description
Transmit interrupt	The acknowledged CAN frame signals a successful transmission to the receiving CAN Controller and triggers the transmit interrupt.
Confirmation to CanIf	CanDrv calls the service CanIf_TxConfirmation(). The parameter CanTxPduId specifies the L-PDU previously sent by Can_Write(). 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 upper layer	Calling of the corresponding upper layer confirmation service <User_TxConfirmation>(). It signals a successful L-SDU transmission to the upper layer.

9.4 Transmit confirmation (polling mode)

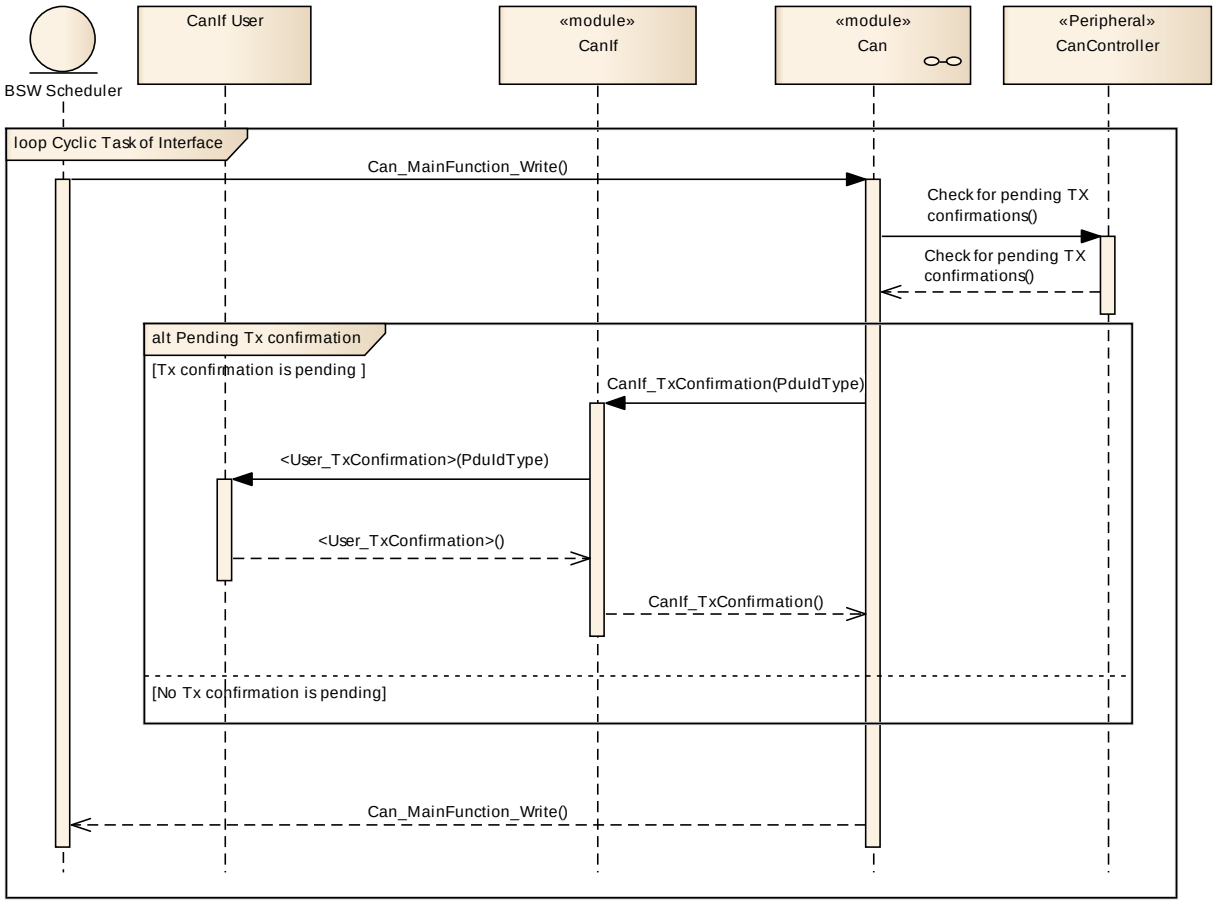

Figure 9.4: Transmit confirmation polling driven

Activity	Description
Cyclic Task CanDrv	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 CanIf	CanDrv calls the service `CanIf_TxConfirmation()`. The parameter `CanTxPduId` specifies the L-PDU previously sent by `Can_Write()`. 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 upper layer	Calling of the corresponding upper layer confirmation service `<User_TxConfirmation>()`. It signals a successful L-SDU transmission to the upper layer.

9.5 Transmit confirmation (with buffering)

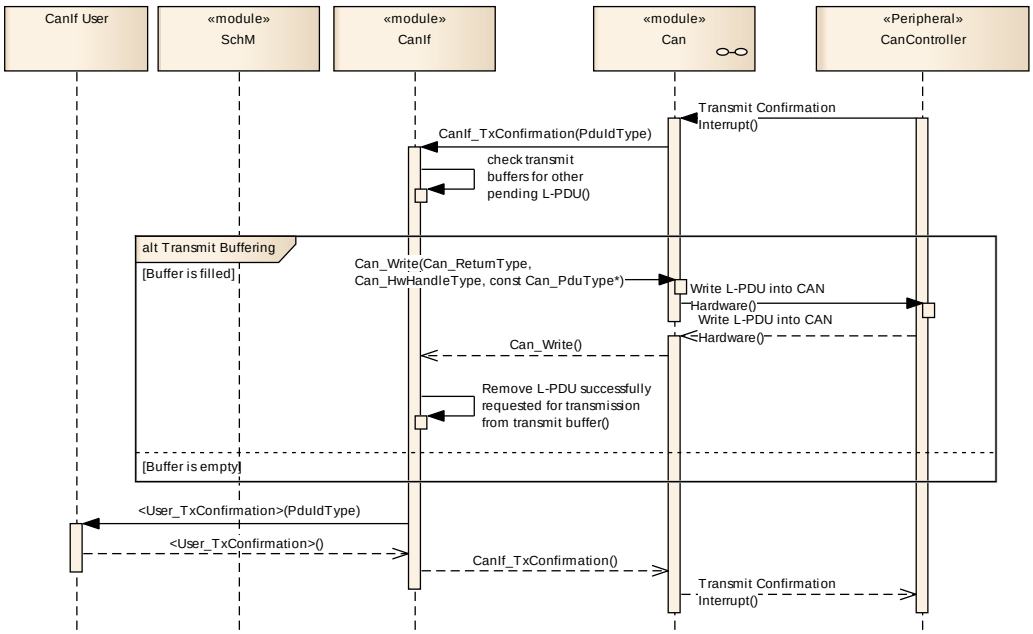

Figure 9.5: Transmit confirmation with buffering

Activity	Description
Transmit interrupt	Acknowledged CAN frame signals successful transmission to receiving CAN Controller and triggers transmit interrupt.
Confirmation to CanIf	CanDrv calls service CanIf_TxConfirmation(). Parameter CanTxPduId specifies the L-PDU previously transmitted by Can_Write(). 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().
Check of transmit buffers	The transmit buffers of CanIf checked, whether a pending L-PDU is stored or not.
Transmit request passed to CanDrv	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 CanIf.
Confirmation to the upper layer	Calling of the corresponding upper layer confirmation service <User_TxConfirmation>(). It signals a successful L-SDU transmission to the upper layer.

9.6 Transmit cancellation (with buffering)

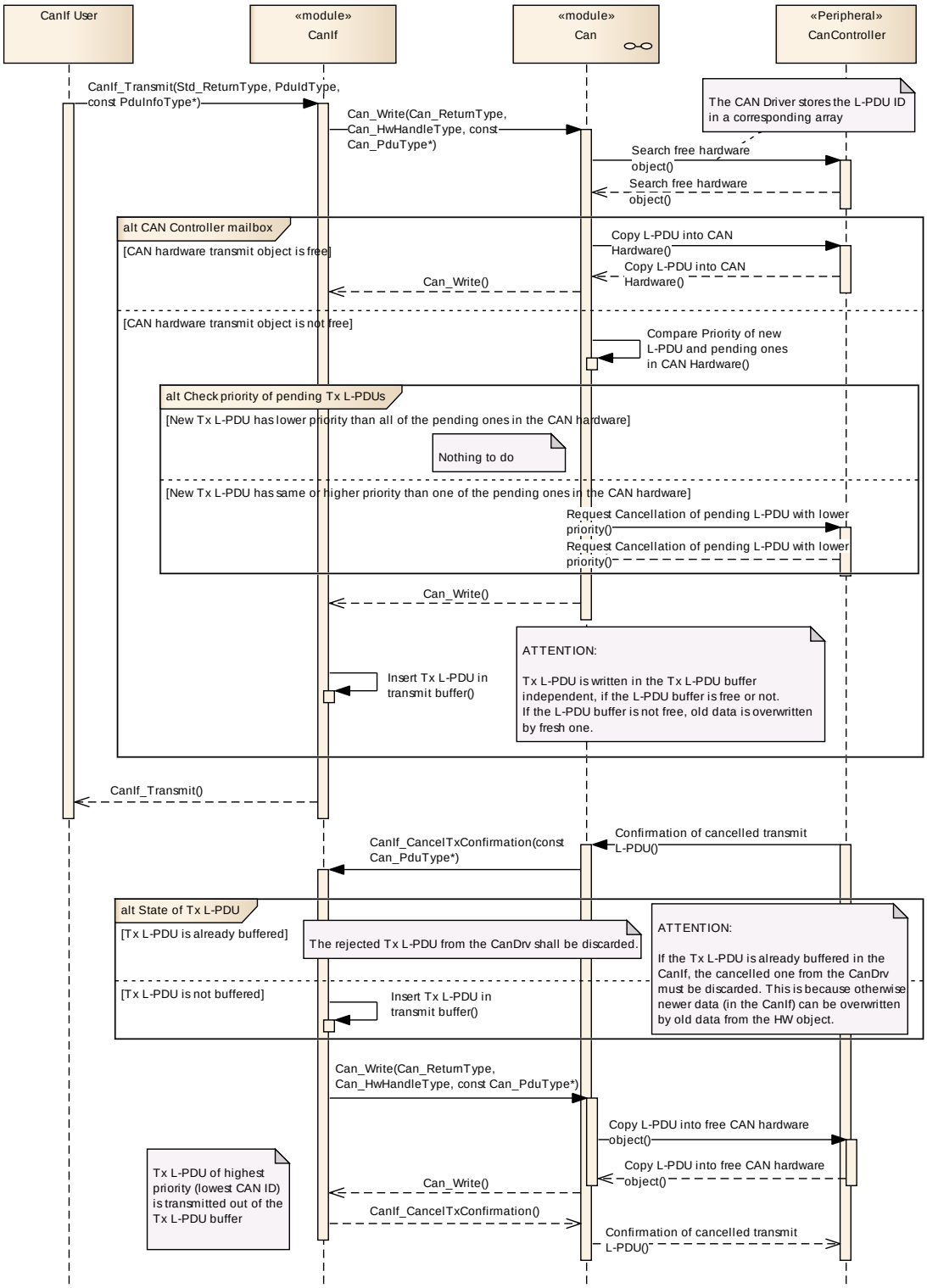

Figure 9.6: Transmit cancellation (with buffering)

Activity	Description
Transmission request	The upper layer initiates a transmit request via the service CanIf_Transmit(). The parameter CanTxPduId identifies the requested L-SDU. 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-SDU related data such as SduLength and *SduDataPtr.
Start transmission	CanIf_Transmit() requests a transmission and calls 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 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 CanIf.
E_BUSY from Can_Write service with transmit abort	If 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 CanIf.
Transmit buffer	CanIf stores the rejected L-PDU in the transmit buffers.
E_OK from CanIf	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 CanIf	CanDrv calls service CanIf_CancelTxConfirmation(). The parameter CanPduPtr specifies the CAN L-PDU successfully aborted by CanDrv. 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().
Check of transmit buffers	The transmit buffer of CanIf checked, whether the L-PDU with the same CanPduPtr->id 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 CanDrv	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 CanIf.
Cancellation confirmation finished	The cancellation confirmation callback returns.

9.7 Transmit cancellation

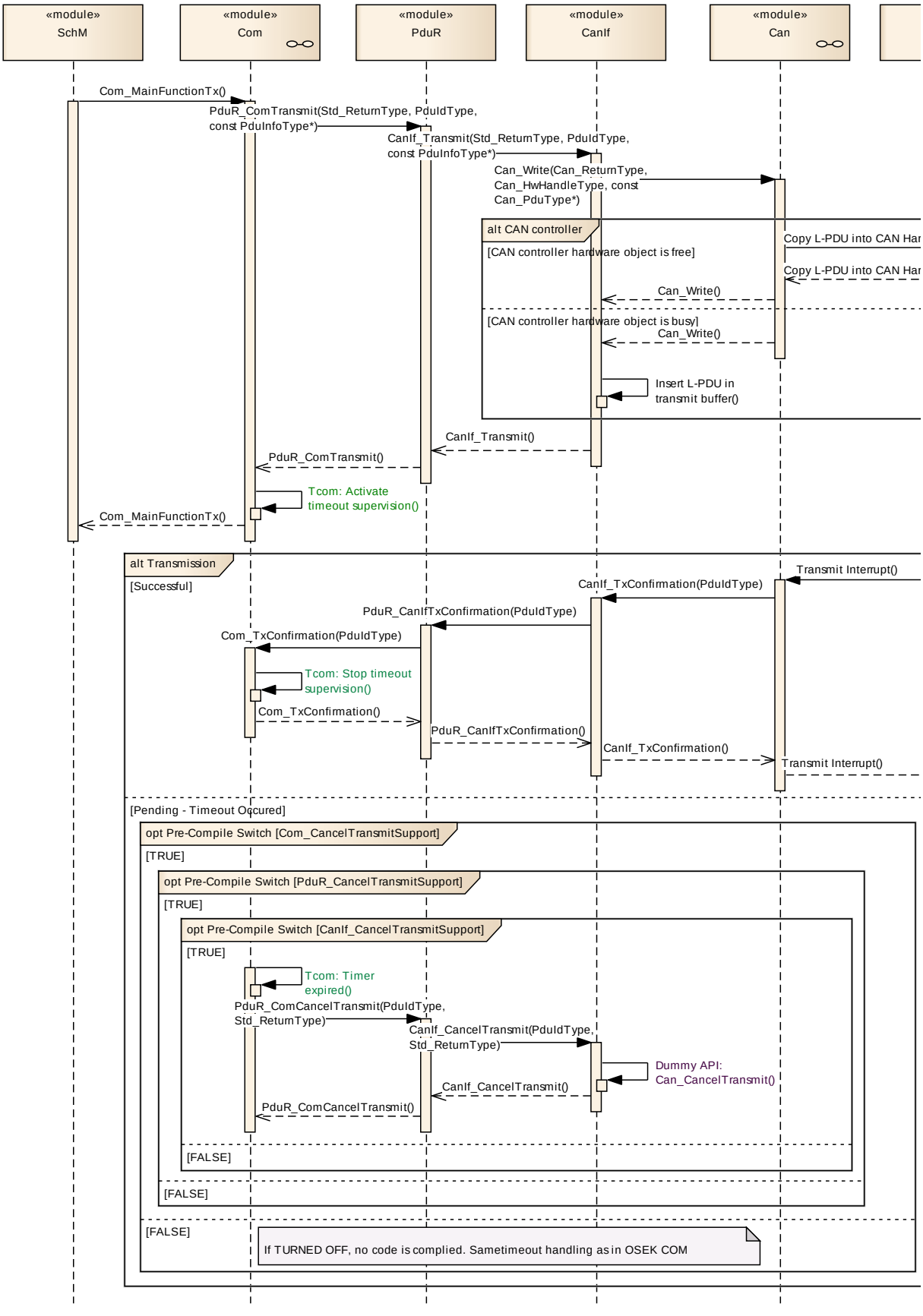

Figure 9.7: Transmit cancellation

Activity	Description
Call of scheduled Function	Com_MainFunctionTx() will be called cyclic by SchM.
Transmission request to PduR	Within cyclic called Com_MainFunctionTx() a transmission request through PduR arises: PduR_ComTransmit()
Transmission request to CanIf	PduR passes the transmit request via CanIf_Transmit() to CanIf. The parameter CanTxPduId identifies the requested L-SDU. 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-SDU related data such as SduLength and *SduDataPtr.
Transmission request to CanDrv	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 CanDrv detects, there are no free hardware objects available, it returns CAN_E_BUSY to 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 CanIf	CanIf_Transmit() returns E_OK to the PduR.
E_OK from PduR	PduR_ComTransmit() returns E_OK to COM.
Starting Timeout supervision	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 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 CanIf	CanDrv calls service CanIf_TxConfirmation(). Parameter CanTxPduId specifies the L-PDU previously sent by Can_Write(). 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 PduR	CanIf calls the service PduR_CanIfTxConfirmation() with the corresponding CanTxPduId.
Confirmation to COM	PduR informs COM 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 COM, is stopped.

Cancellation confirmation notification:

Activity	Description
Transmit cancellation to PduR	If `Com_CancelTransmitSupport`, `PduR_CancelTransmitSupport` and `CanIf_CancelTransmitSupport` are activated, the API `PduR_ComCancelTransmit()` is called by COM with the corresponding parameter `ComTxPduId` e.g. after a timer has been expired.
Transmit cancellation to CanIf	If `PduR` passes the transmit cancellation via the service `CanIf_CancelTransmit()` to `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 `PduR`.
`E_NOT_OK` from `PduR_ComCancelTransmit`	PduR returns `E_NOT_OK` to COM.

9.8 Receive indication (interrupt mode)

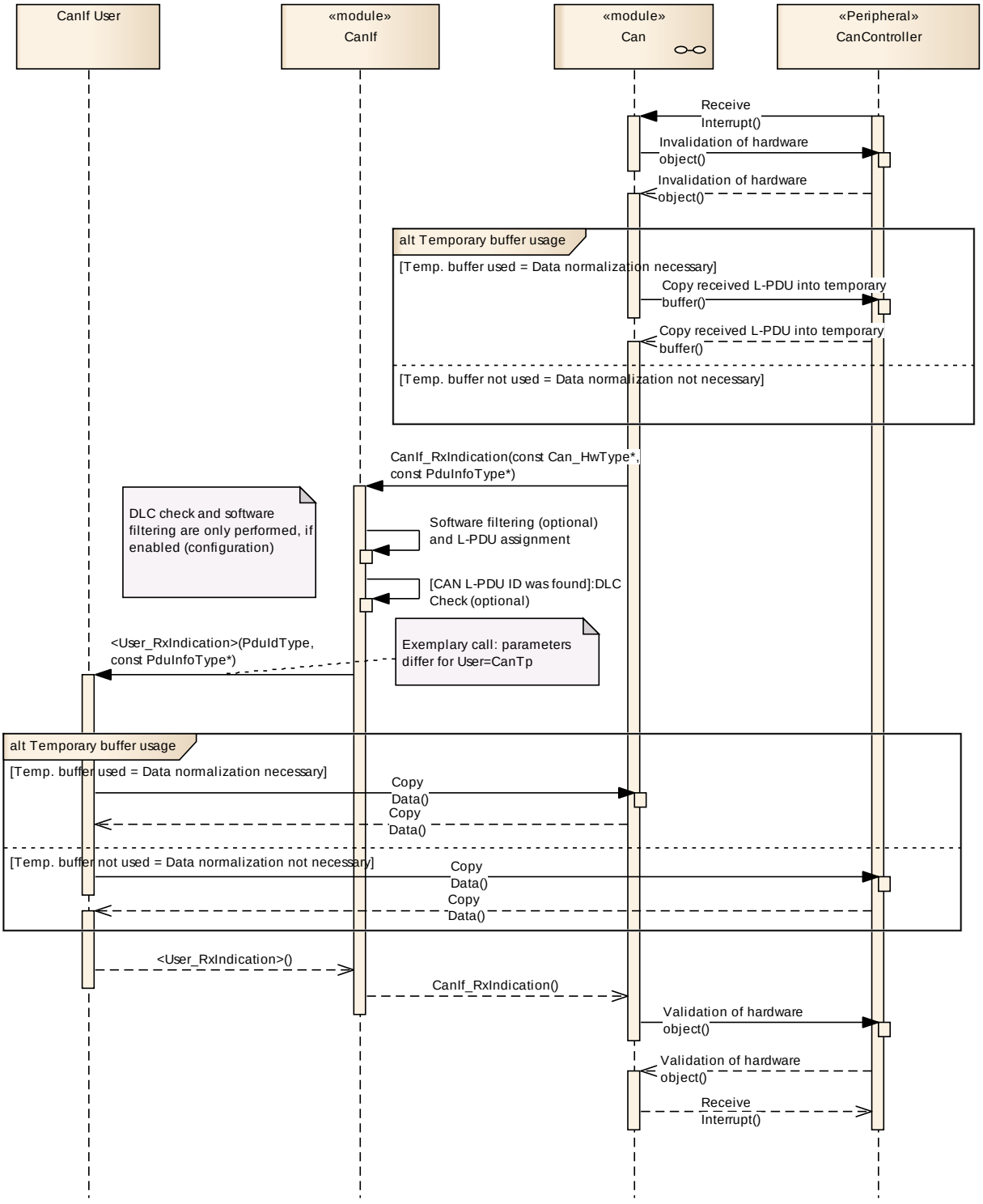

Figure 9.8: Receive indication interrupt driven

Activity	Description
Receive Interrupt	The `CAN Controller` indicates a successful reception and triggers a receive interrupt.

Invalidation of CAN hardware object, provide CPU access to CAN mailbox	The CPU (CanDrv) get exclusive access rights to the CAN mailbox or at least to the corresponding hardware object, where new data were received.
Buffering, normalizing	The L-PDU is normalized and is buffered in the temporary buffer located in CanDrv. Each CanDrv owns such a temporary buffer for every Physical Channel only if normalizing of the data is necessary.
Indication to CanIf	The reception is indicated to CanIf 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 CanIf by PduInfoPtr->SduDataPtr.
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 RxPduId specifies the L-SDU, 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)

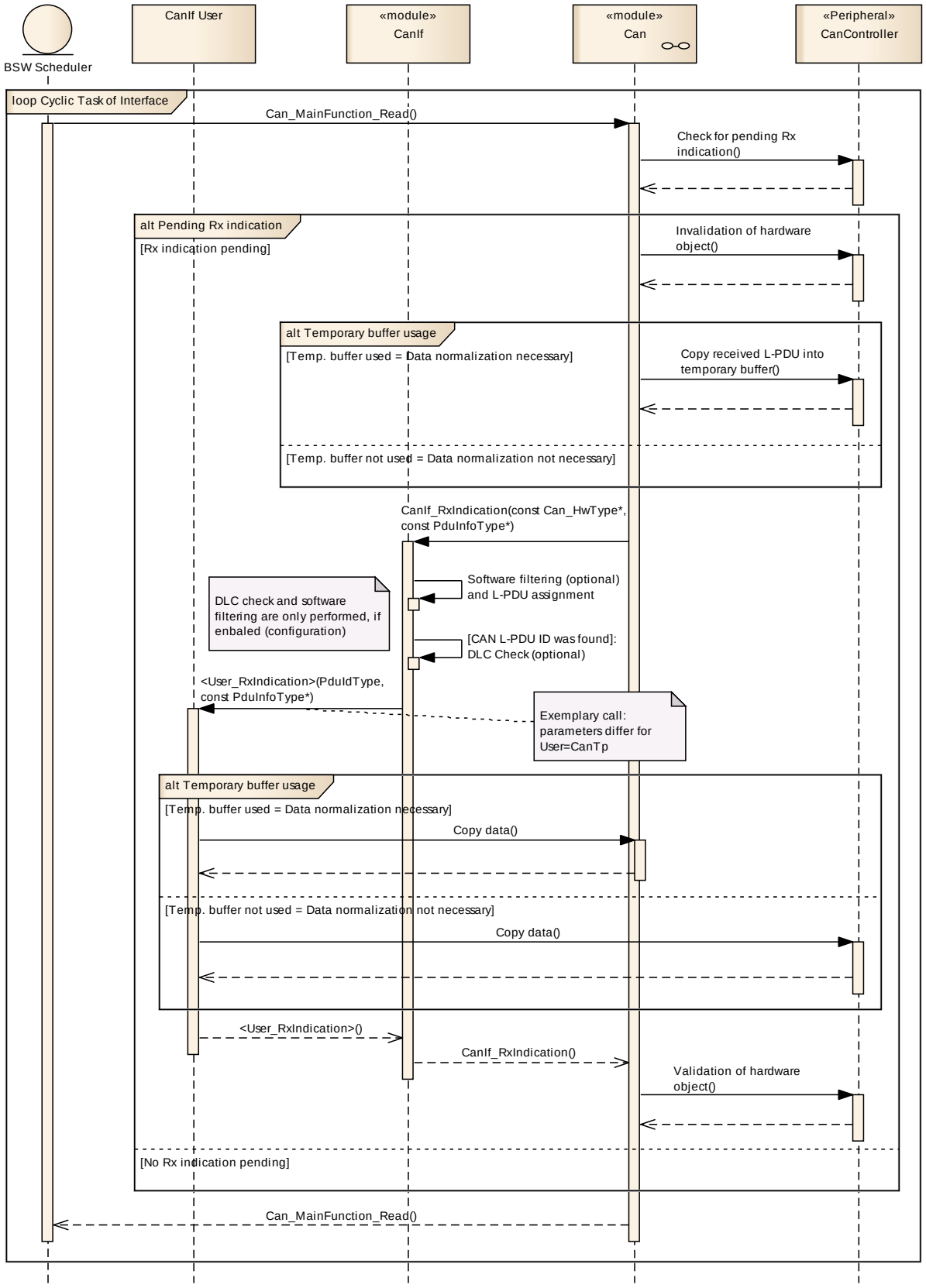

Figure 9.9: Receive indication polling driven

Activity	Description
Cyclic Task CanDrv	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 (CanDrv) 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-PDU is normalized and is buffered in the temporary buffer located in CanDrv. Each CanDrv owns such a temporary buffer for every Physical Channel only if normalizing of the data is necessary.
Indication to CanIf	The reception is indicated to CanIf 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 CanIf by PduInfoPtr->SduDataPtr.
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 RxPduId specifies the L-SDU, 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

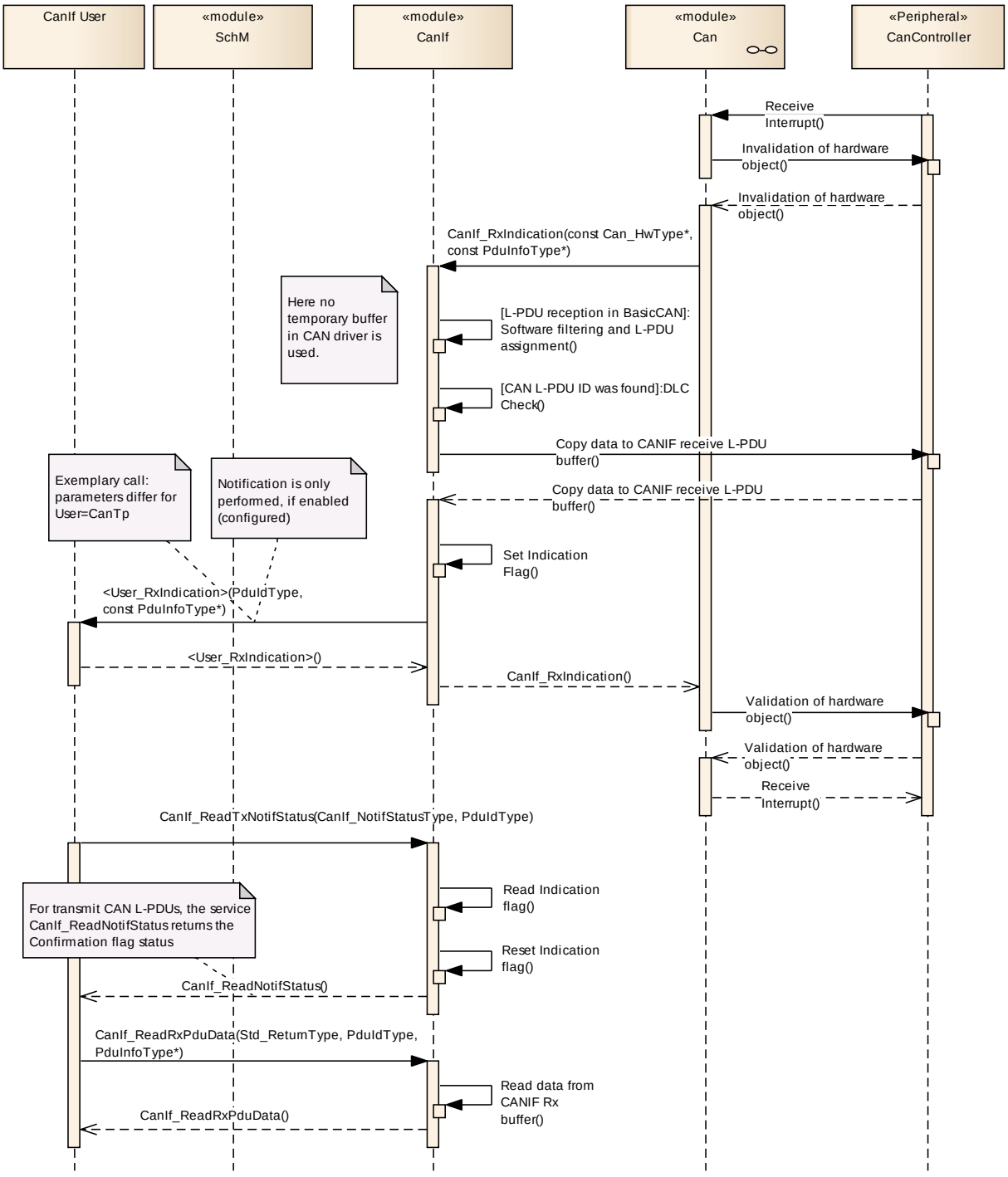

Figure 9.10: Read received data

Activity	Description
Receive Interrupt	The CAN Controller indicates a successful reception and triggers a receive interrupt.

Invalidation of CAN hardware object, provide CPU access to CAN mailbox	The CPU (CanDrv) get exclusive access rights to the CAN mailbox or at least to the corresponding hardware object, where new data were received.
Buffering, normalizing	The L-PDU is normalized and is buffered in the temporary buffer located in CanDrv. Each CanDrv owns such a temporary buffer for every Physical Channel only if normalizing of the data is necessary.
Indication to CanIf	The reception is indicated to CanIf 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 CanIf by PduInfoPtr->SduDataPtr.
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 CanIf. During access the CAN hardware buffers must be unlocked for CPU access/locked for CAN Controller access.
Indication Flag	Set indication status flag for the received L-PDU in CanIf.
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 RxPduId specifies the L-SDU, 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_ReadRxPduData().
Read CanIf Rx buffer	CanIf_ReadRxPduData() reads the data from CanIf Rx buffer.
E_OK from CanIf	If CanIf_ReadRxPduData() was successful, the request returns E_OK with valid PduInfoPtr.

9.11 Start CAN network

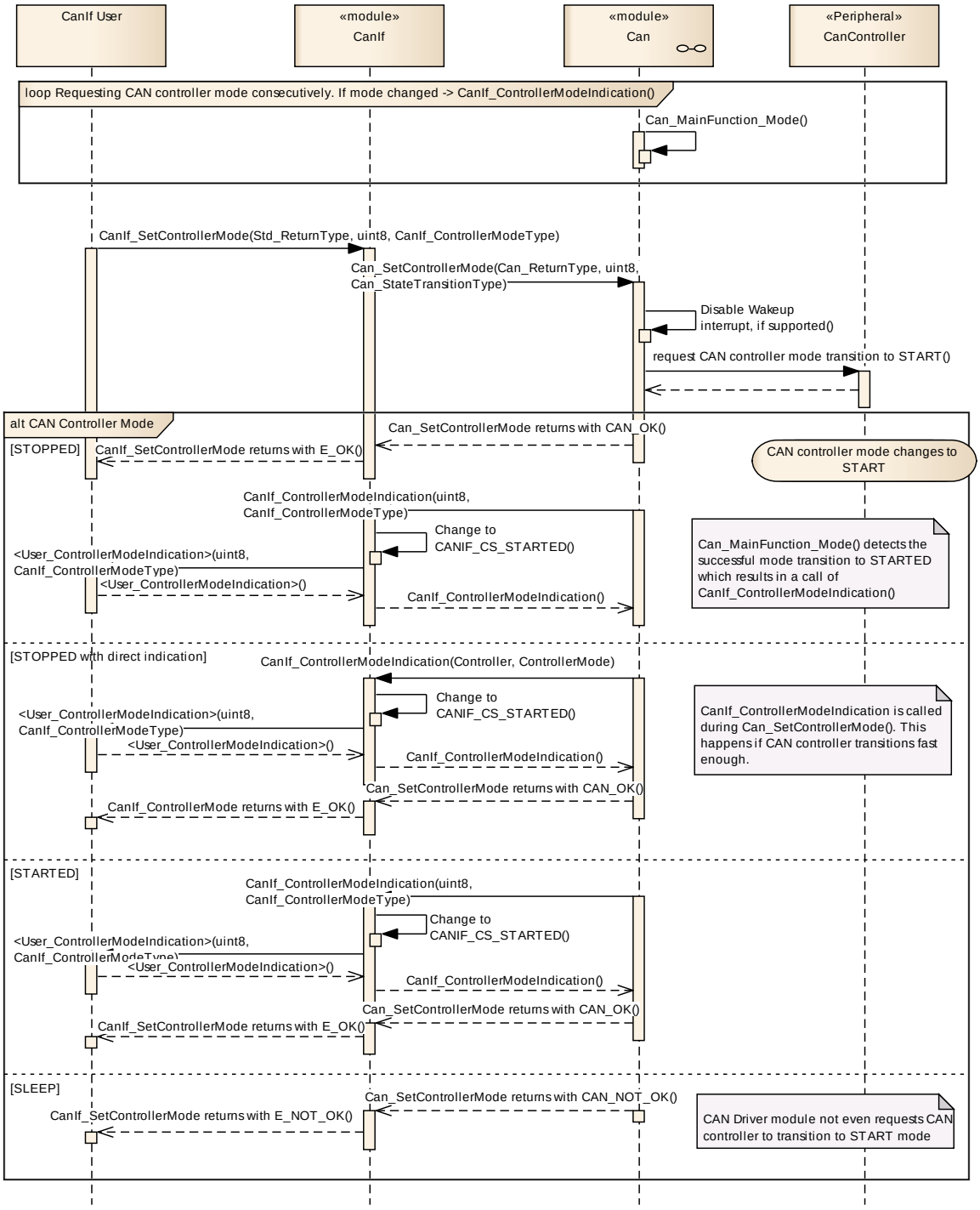

Figure 9.11: Start CAN network

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

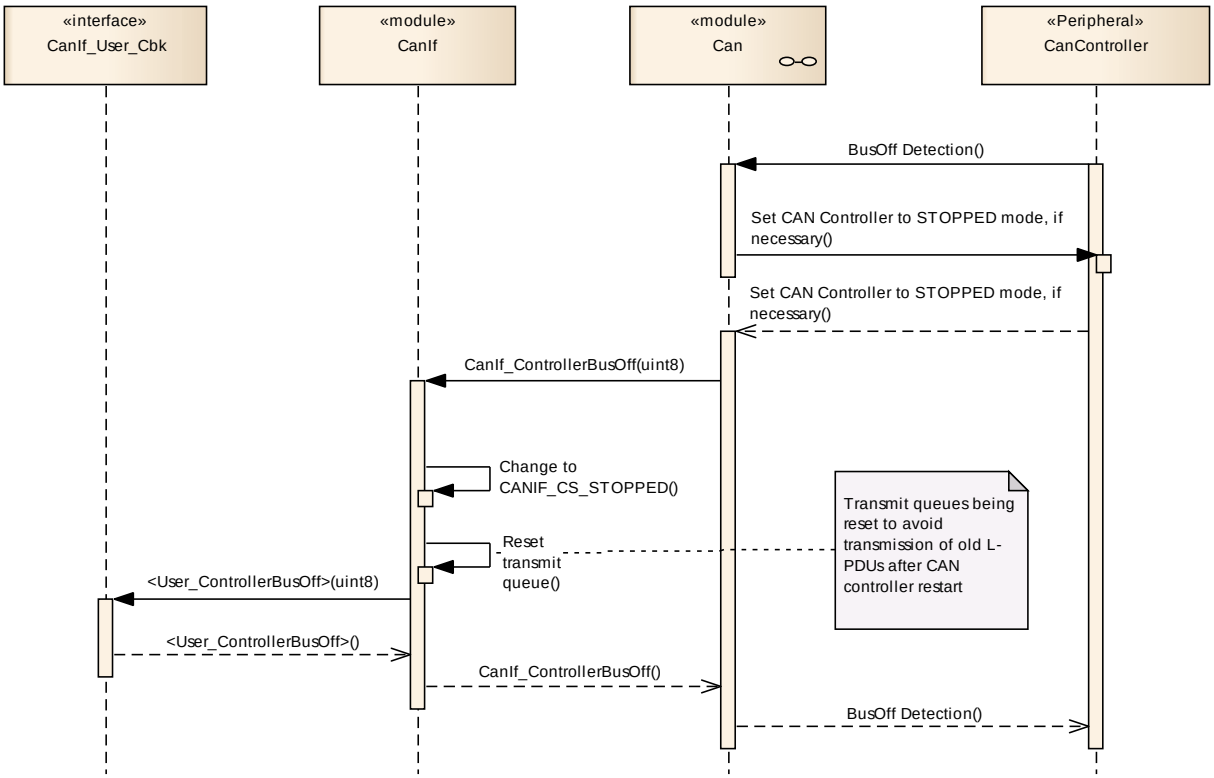

Figure 9.12: BusOff notification

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

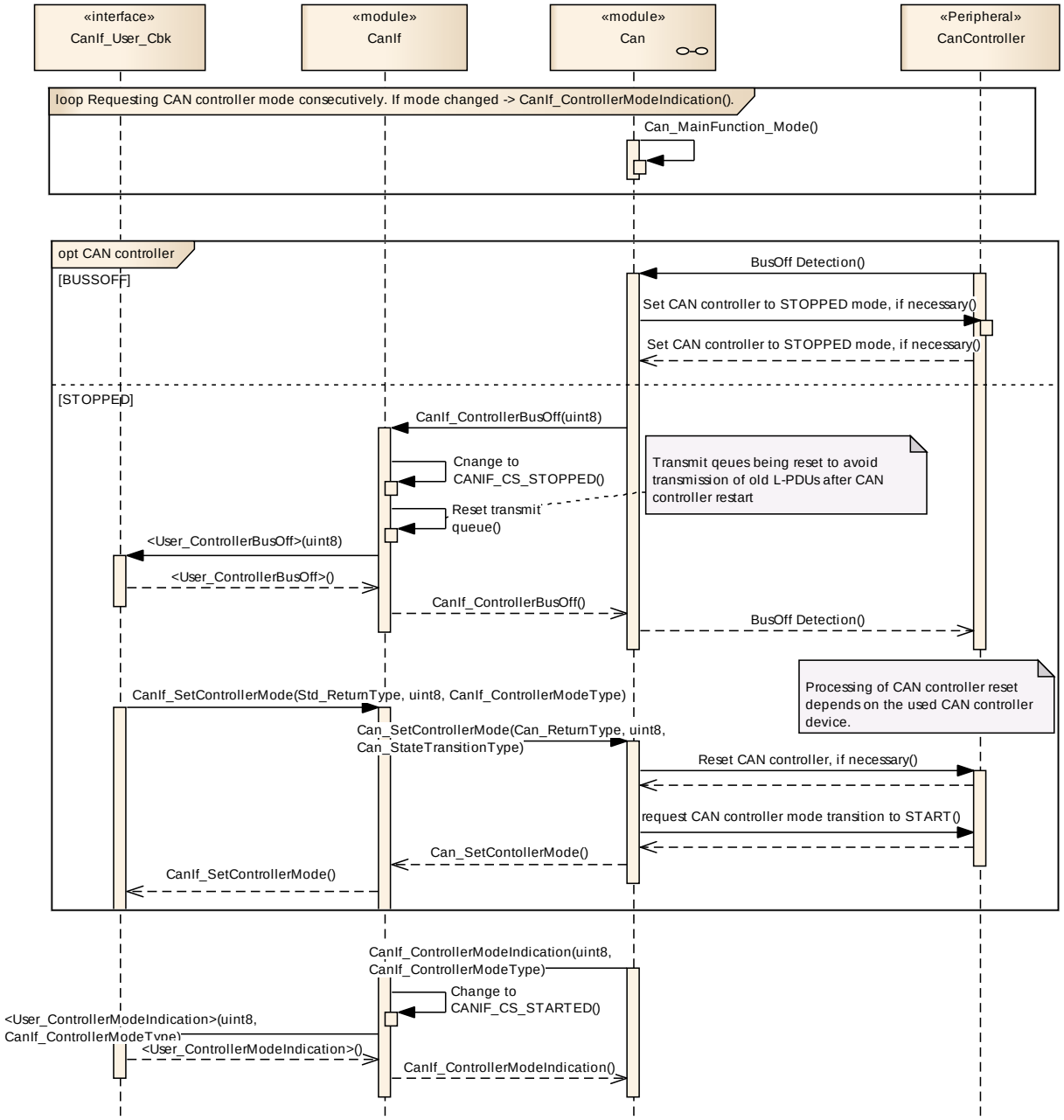

Figure 9.13: BusOff recovery

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 CanIf	BusOff is notified to the CanIf by calling of CanIf_ControllerBusOff(). The transmit buffers inside 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).
CAN controller started	CanDrv informs CanIf about the successful start by calling CanIf_ControllerModeIndication(). CanIf changes mode to CANIF_CS_STARTED and informs in turn upper layers about the mode change.

10 Configuration specification

In general, this chapter defines configuration parameters and their clustering into containers. In order to support the specification section 10.1 describes fundamentals. It also specifies a template (table) you shall use for the parameter specification. We intend to leave section 10.1 in the specification to guarantee comprehension.

section 10.2 specifies the structure (containers) and the parameters of the CanIf.

10.1 How to read this chapter

For details refer to the [9, chapter 10.1 "Introduction to configuration specification" in SWS_BSWGeneral]

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.

[SWS_CANIF_00104] ⌈ 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`. ⌋*(SRS_CAN_01015)*

[SWS_CANIF_00066] ⌈ The CanIf has access to the `CanDrv` configuration data. All public CanDrv configuration data are described in [1, Specification of CAN Driver]. ⌋

[SWS_CANIF_00132] ⌈ These dependencies between CanDrv and CanIf configuration must be provided at configuration time by the configuration tools. ⌋

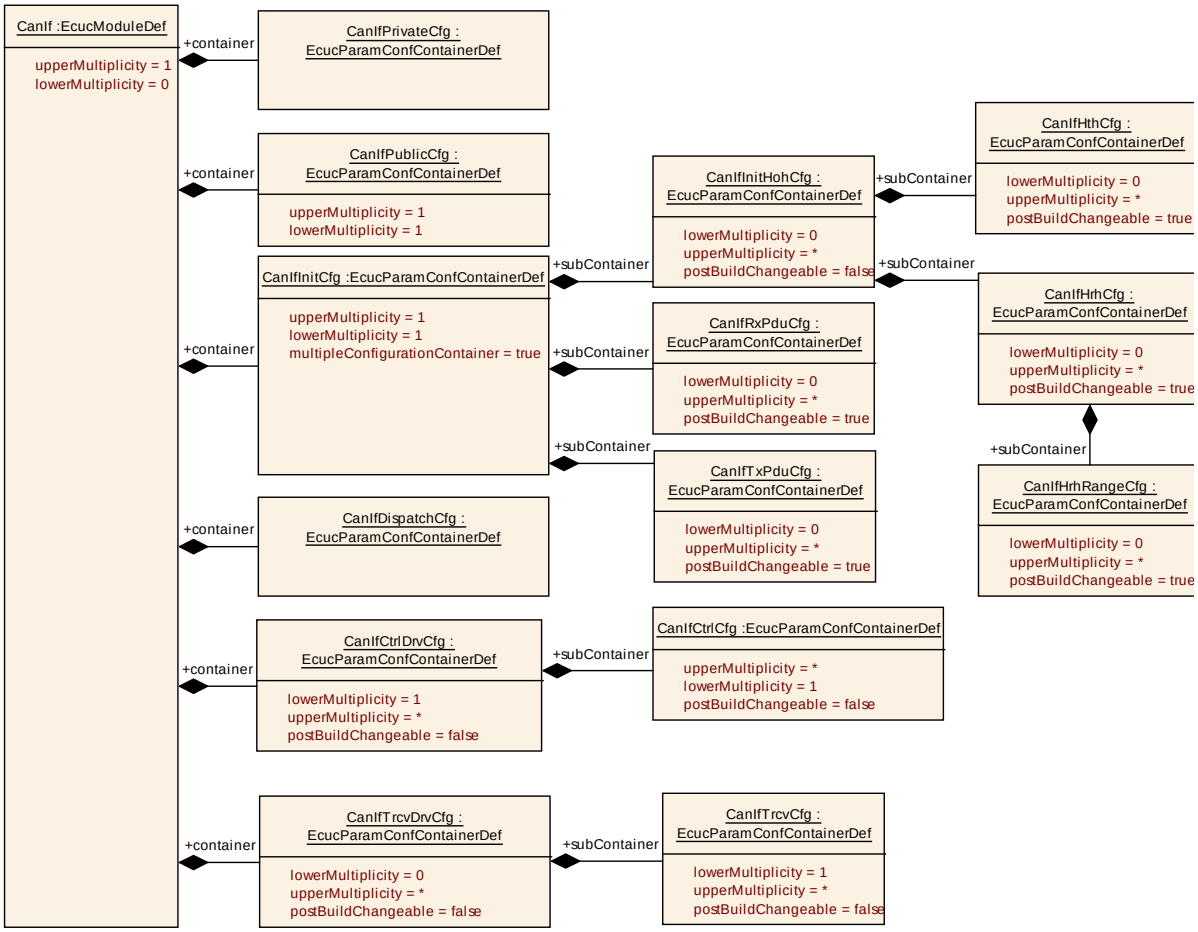

Figure 10.1: Overview about CAN Interface configuration containers

Variants

[SWS_CANIF_00460] ⌈ Variant 1: Only pre compile time parameters. ⌋*(SRS_BSW_00344)*

[SWS_CANIF_00461] ⌈ Variant 2: Mix of pre compile- and link time parameters. ⌋*(SRS_BSW_00344)*

[SWS_CANIF_00462] ⌈ Variant 3:Mix of pre compile-, link time and post build time parameters. ⌋*(SRS_BSW_00344, SRS_BSW_00404, SRS_BSW_00342)*

CanIf

[ECUC_CanIf_00244] belongs to the table below. The generated Artifact is faulty.

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.

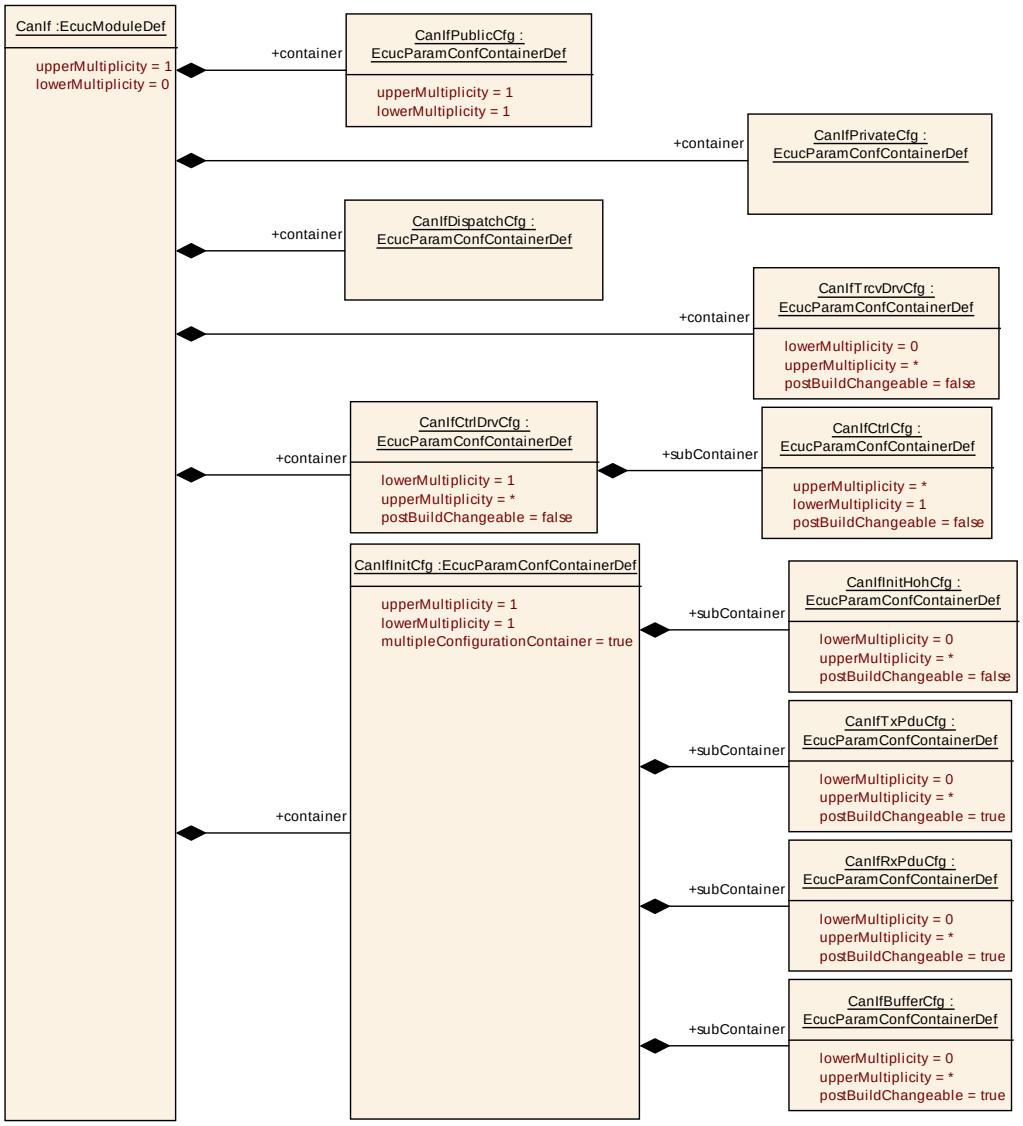

Figure 10.2: AR_EcucDef_CanIf

CanIfPrivateCfg

SWS Item	[ECUC_CanIf_00245]
Container Name	CanIfPrivateCfg {CanInterfacePrivateConfiguration}
Description	This container contains the private configuration (parameters) of the CAN Interface.
Configuration Parameters	

Name	CanIfFixedBuffer [ECUC_CanIf_00827]		
Description	This parameter defines if the buffer element length shall be fixed to 8 Bytes. TRUE: Buffer element length is fixed to 8 Bytes. FALSE: Buffer element length depends on the size of the referencing PDUs.		
Multiplicity	0..1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local		

Name	CanIfPrivateDlcCheck {CANIF_PRIVATE_DLC_CHECK} [ECUC_CanIf_00617]		
Description	Selects whether the DLC check is supported. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	true		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local		

Name	CanIfPrivateSoftwareFilterType {CANIF_PRIVATE_SOFTWARE_FILTER_TYPE} [ECUC_CanIf_00619]		
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.	
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local 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.		

Name	CanIfSupportTTCAN [ECUC_CanIf_00675]		
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		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Included Containers		
Container Name	**Multiplicity**	**Scope / Depedency**
CanIfTTGeneral	0..1	CanIfTTGeneral is specified in the SWS TTCAN Interface and defines if and in which way TTCAN is supported. This container is only included and valid if TTCAN is supported by the controller, enabled (see CanIfSupportTTCAN, ECUC_CanIf_00675), and used.

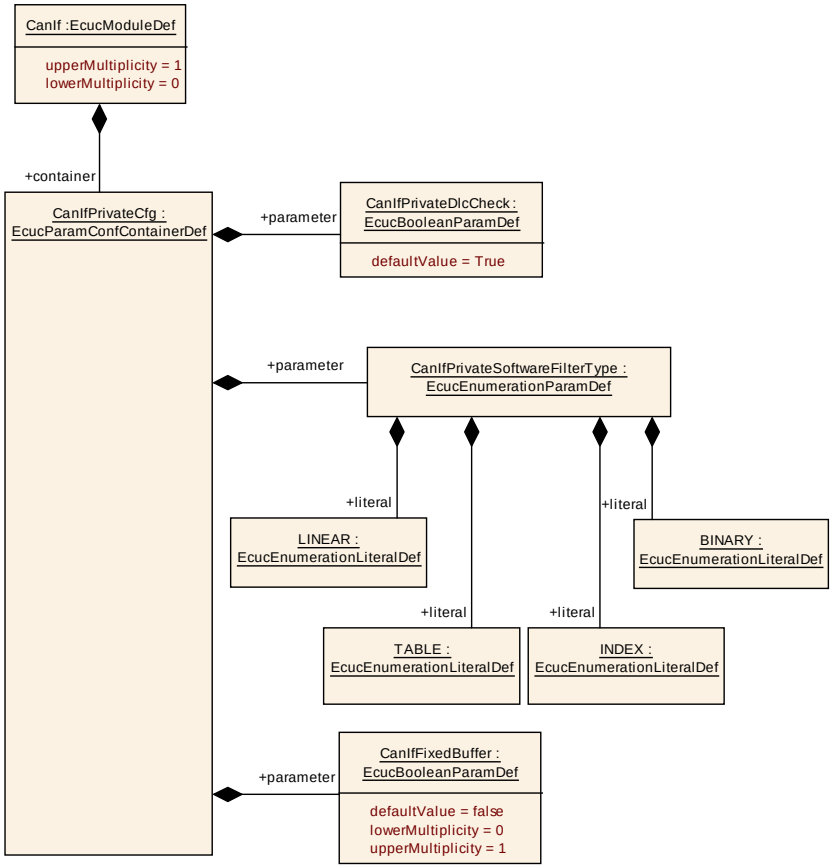

Figure 10.3: AR_EcucDef_CanIfPrivateCfg

CanIfPublicCfg

SWS Item	[ECUC_CanIf_00246]
Container Name	CanIfPublicCfg {CanInterfacePublicConfiguration}
Description	This container contains the public configuration (parameters) of the CAN Interface.
Configuration Parameters	

Name	CanIfMetaDataSupport {CANIF_META_DATA_SUPPORT} [ECUC_CanIf_00824]		
Description	Enable support for dynamic ID handling using L-SDU MetaData.		
Multiplicity	0..1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfPublicCancelTransmitSupport {CANIF_PUBLIC_CANCEL_TRANSMIT_SUPPORT} [ECUC_CanIf_00522]		
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			
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfPublicCddHeaderFile {CANIF_PUBLIC_CDD_HEADERFILE} [ECUC_CanIf_00671]		
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			
Length	1–32		
Regular Expression			
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfPublicChangeBaudrateSupport {CANIF_PUBLIC_CHANGE_BAUDRATE_SUPPORT} [ECUC_CanIf_00785] (Obsolete)		
Description	Configuration parameter to enable/disable the API to change the baudrate of a CAN controller. True: Enabled False: Disabled Please note that the CanIf_ChangeBaudrate API and this parameter are deprecated and will be removed in future. **Tags:** atp.Status=obsolete atp.StatusRevisionEnd=4.1.1		
Multiplicity	0..1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfPublicDevErrorDetect {CANIF_PUBLIC_DEV_ERROR_DETECT} [ECUC_CanIf_00614]		
Description	Enables and disables the development error detection and notification mechanism. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	true		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local		

Name	CanIfPublicHandleTypeEnum {CANIF_PUBLIC_HANDLE_TYPE_ENUM} [ECUC_CanIf_00742]		
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		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU dependency: Can_HwHandleType		

Name	CanIfPublicIcomSupport {CANIF_PUBLIC_ICOM_SUPPORT} [ECUC_CanIf_00839]		
Description	Selects support of Pretended Network features in CanIf. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfPublicMultipleDrvSupport {CANIF_PUBLIC_MULTIPLE_DRV_SUPPORT} [ECUC_CanIf_00612]		
Description	Selects support for multiple CAN Drivers. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	true		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfPublicNumberOfCanHwUnits {CANIF_PUBLIC_NUMBER_OF_CAN_HW_UNITS} [ECUC_CanIf_00615] (Obsolete)		
Description	This parameter is set to obsolete and will be removed. It is not required because the same information is available by checking the actual multiplicity of CanIfCtrlDrvCfg. Old description: Number of served CAN hardware units. **Tags:** atp.Status=obsolete atp.StatusComment=Not required because the same information is available by checking the actual multiplicity of CanIfCtrlDrvCfg. atp.StatusRevisionBegin=4.1.1		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	1 .. 255		
Default Value	1		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfPublicPnSupport {CANIF_PUBLIC_PN_SUPPORT} [ECUC_CanIf_00772]		

Description	Selects support of Partial Network features in CanIf.
	`True: Enabled`
	`False: Disabled`		

Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfPublicReadRxPduDataApi {CANIF_PUBLIC_READRXPDU_DATA_API} [ECUC_CanIf_00607]		
Description	Enables / Disables the API CanIf_ReadRxPduData() for reading received L-SDU data. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfPublicReadRxPduNotifyStatusApi {CANIF_PUBLIC_READRXPDU_NOTIFY_STATUS_API} [ECUC_CanIf_00608]		
Description	Enables and disables the API for reading the notification status of receive L-PDUs. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfPublicReadTxPduNotifyStatusApi {CANIF_PUBLIC_READTXPDU_NOTIFY_STATUS_API} [ECUC_CanIf_00609]		
Description	Enables and disables the API for reading the notification status of transmit L-PDUs. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfPublicSetDynamicTxIdApi {CANIF_PUBLIC_SETDYNAMICTXID_API} [ECUC_CanIf_00610]		
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		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfPublicTxBuffering {CANIF_PUBLIC_TX_BUFFERING} [ECUC_CanIf_00618]		
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		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfPublicTxConfirmPollingSupport {CANIF_PUBLIC_TXCONFIRM_POLLING_SUPPORT} [ECUC_CanIf_00733]		
Description	Configuration parameter to enable/disable the API to poll for Tx Confirmation state.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value			

Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local dependency: CAN State Manager module		

Name	CanIfPublicVersionInfoApi {CANIF_PUBLIC_VERSION_INFO_API} [ECUC_CanIf_00613]		
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		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local		

Name	CanIfPublicWakeupCheckValidByNM {CANIF_PUBLIC_WAKEUP_CHECK_VALID_BY_NM} [ECUC_CanIf_00741]		
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		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU dependency: CANIF_PUBLIC_WAKEUP_CHECK_VALID_API		

Name	CanIfPublicWakeupCheckValidSupport {CANIF_PUBLIC_WAKEUP_CHECK_VALIDATION_SUPPORT} [ECUC_CanIf_00611]		
Description	Selects support for wake up validation True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		

Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfSetBaudrateApi {CANIF_SET_BAUDRATE_API} [ECUC_CanIf_00838]		
Description	Configuration parameter to enable/disable the CanIf_SetBaudrate API to change the baud rate of a CAN Controller. If this parameter is set to true the CanIf_SetBaudrate API shall be supported. Otherwise the API is not supported.		
Multiplicity	0..1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfTxOfflineActiveSupport {CANIF_TX_OFFLINE_ACTIVE_PUBLIC_SUPPORT} [ECUC_CanIf_00837]		
Description	Determines wether TxOffLineActive feature (see SWS_CANIF_00072) is supported by CanIf. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

No Included Containers

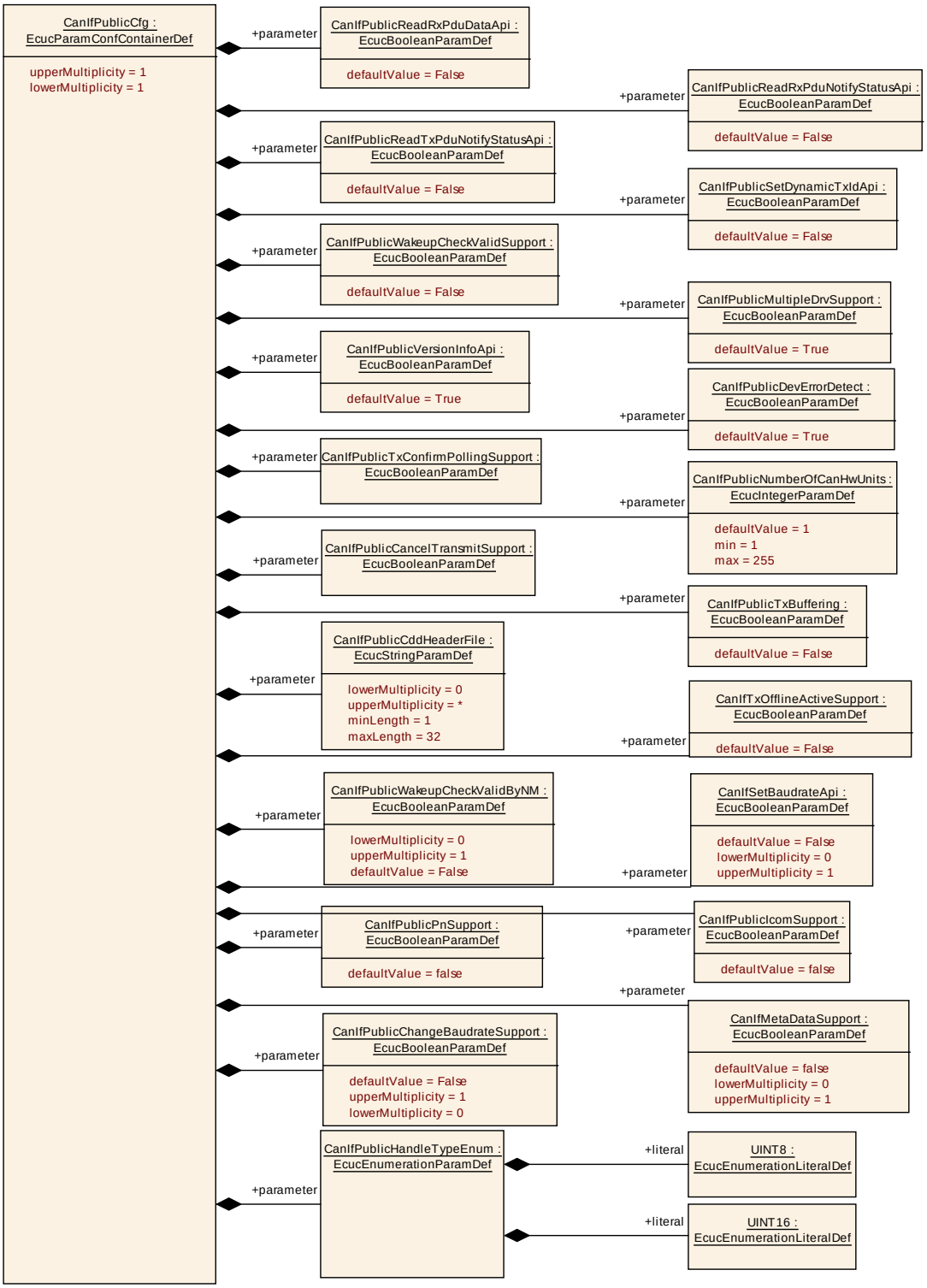

Figure 10.4: AR_EcucDef_CanIfPublicCfg

CanIfInitCfg

SWS Item	[ECUC_CanIf_00247]
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	

Name	CanIfInitCfgSet {CANIF_INIT_CONFIGSET} [ECUC_CanIf_00623]		
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			
Length	1–32		
Regular Expression			
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

Name	CanIfMaxBufferSize [ECUC_CanIf_00828]		
Description	Maximum total size of all Tx buffers. This parameter is needed only in case of post-build loadable implementation using static memory allocation.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 18446744073709551615		
Default Value			
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: local		

Name	CanIfMaxRxPduCfg [ECUC_CanIf_00830]		
Description	Maximum number of Pdus. This parameter is needed only in case of post-build loadable implementation using static memory allocation.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 18446744073709551615		
Default Value			

Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: local		

Name	CanIfMaxTxPduCfg [ECUC_CanIf_00829]		
Description	Maximum number of Pdus. This parameter is needed only in case of post-build loadable implementation using static memory allocation.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 18446744073709551615		
Default Value			
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: local		

Included Containers

Container Name	Multiplicity	Scope / Depedency
CanIfBufferCfg	0..*	This container contains the Txbuffer configuration. Multiple buffers with different sizes could be configured. If CanIfBufferSize (ECUC_CanIf_00834) 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.

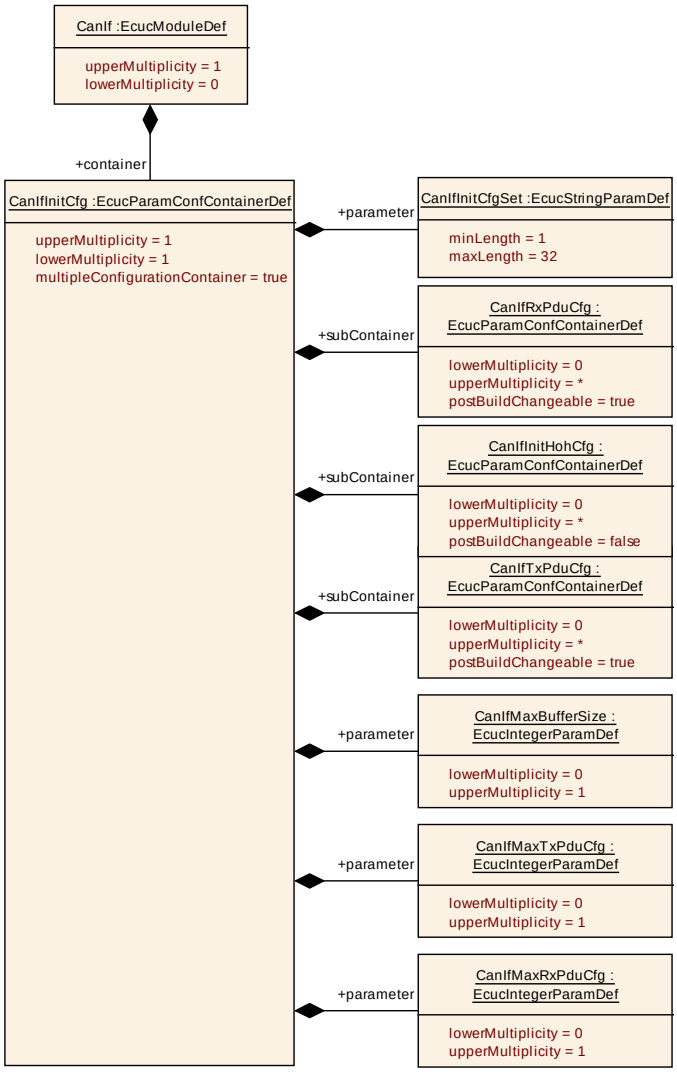

Figure 10.5: AR_EcucDef_CanIfInitCfg

CanIfTxPduCfg

SWS Item	[ECUC_CanIf_00248]
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. **Attributes:** postBuildChangeable=true
Configuration Parameters	

Name	CanIfTxPduBufferRef {CANIF_TX_PDU_BUFFER_REF} [ECUC_CanIf_00831]		
Description	Configurable reference to a CanIf buffer configuration.		
Multiplicity	1		
Type	Reference to CanIfBufferCfg		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

Name	CanIfTxPduCanId {CANIF_TXPDU_CANID} [ECUC_CanIf_00592]		
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 The CAN Identifier may be omitted for dynamic transmit L-PDUs.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 536870911		
Default Value			
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

Name	CanIfTxPduCanIdMask {CANIF_TXPDU_CANID_MASK} [ECUC_CanIf_00823]		
Description	Identifier mask which denotes relevant bits in the CAN Identifier. This parameter may be used to keep parts of the CAN Identifier of dynamic transmit L-PDUs static. Range: 11 bits for Standard CAN Identifier, 29 bits for Extended CAN Identifier.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 536870911		
Default Value	536870911		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

Name	CanIfTxPduCanIdType {CANIF_TXPDU_CANIDTYPE} [ECUC_CanIf_00590]	
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	CAN frame with extended identifier (29 bits)

	EXTENDED_FD_CAN	CAN FD frame with extended identifier (29 bits)	
	STANDARD_CAN	CAN frame with standard identifier (11 bits)	
	STANDARD_FD_CAN	CAN FD frame with standard identifier (11 bits)	
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

Name	CanIfTxPduDlc {CANIF_TXPDU_DLC} [ECUC_CanIf_00594] (Obsolete. Use CanIfTxPduRef, CanIfFixedBuffer instead.)		
Description	This parameter is set to obsolete and will be removed. It was replaced by a combination of the size of the global PDU (referenced via CanIfTxPduRef) and the parameter CanIfFixedBuffer. Old 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. **Tags:** atp.Status=obsolete atp.StatusComment=Replaced by a combination of the size of the global PDU (referenced via CanIfTxPduRef) and the parameter CanIf FixedBuffer. atp.StatusRevisionBegin=4.1.1		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 8		
Default Value			
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

Name	CanIfTxPduId {CANIF_TXPDU_ID} [ECUC_CanIf_00591]		
Description	ECU wide unique, symbolic handle for transmit CAN L-SDU. Range: 0..max. number of CantTxPduIds		
Multiplicity	1		
Type	EcucIntegerParamDef (Symbolic Name generated for this parameter)		
Range	0 .. 4294967295		
Default Value			
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfTxPduPnFilterPdu {CANIF_TXPDU_PNFILTERPDU} [ECUC_CanIf_00773]		
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		
Configuration Class	**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: This parameter shall only be configurable if CanIfPublicPnSupport equals True.		

Name	CanIfTxPduReadNotifyStatus {CANIF_TXPDU_READ_NOTIFYSTATUS} [ECUC_CanIf_00589]		
Description	Enables and disables transmit confirmation for each transmit CAN L-SDU for reading its notification status. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**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: CANIF_READTXPDU_NOTIFY_STATUS_API must be enabled.		

Name	CanIfTxPduRef {CANIF_TXPDU_REF} [ECUC_CanIf_00603]		
Description	Reference to the "global" Pdu structure to allow harmonization of handle IDs in the COM-Stack.		
Multiplicity	1		
Type	Reference to Pdu		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

Name	CanIfTxPduType {CANIF_TXPDU_TYPE} [ECUC_CanIf_00593]	
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.	
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

Name	CanIfTxPduUserTxConfirmationName {CANIF_TXPDU_USERTXCONFIRMATION_NAME} [ECUC_CanIf_00528]		
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, J1939NM 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			
Length	1–32		
Regular Expression			
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfTxPduUserTxConfirmationUL {CANIF_TXPDU_USERTXCONFIRMATION_UL} [ECUC_CanIf_00527]	
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 Driver
	J1939NM	J1939Nm
	J1939TP	J1939Tp
	PDUR	PDU Router

	XCP	Extended Calibration Protocol	
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: ECU		

Included Containers		
Container Name	**Multiplicity**	**Scope / Depedency**
CanIfTTTxFrame Triggering	0..1	CanIfTTTxFrameTriggering is specified in the SWS TTCAN Interface and defines Frame trigger for TTCAN transmission. This container is only included and valid if TTCAN is supported by the controller, enabled (see CanIfSupportTTCAN, ECUC_CanIf_00675), and a joblist is used.

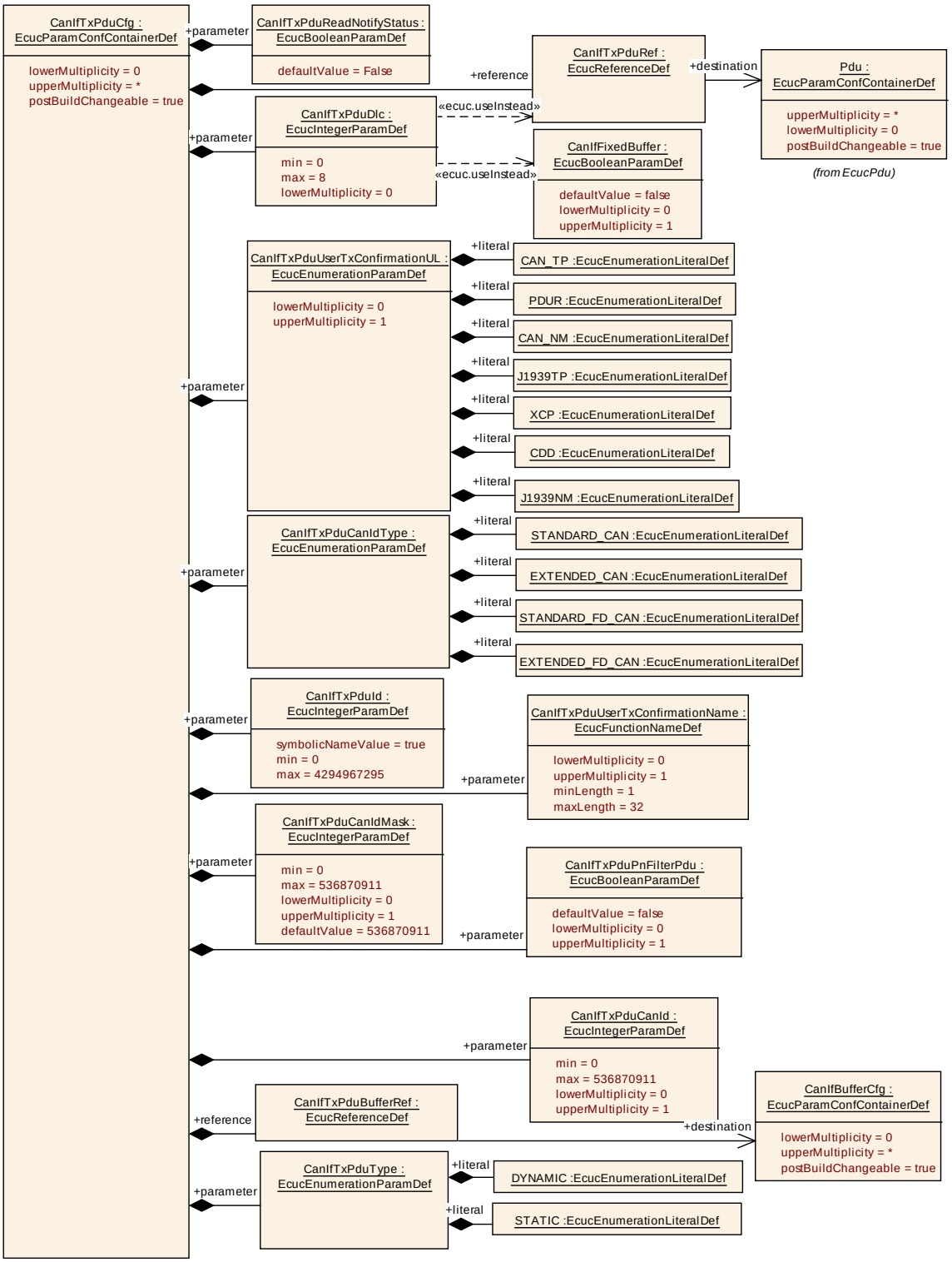

Figure 10.6: AR_EcucDef_CanIfTxPduCfg

CanIfRxPduCfg

SWS Item	[ECUC_CanIf_00249]
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. **Attributes:** postBuildChangeable=true
Configuration Parameters	

Name	CanIfRxPduCanId {CANIF_RXPDU_CANID} [ECUC_CanIf_00598]		
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			
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

Name	CanIfRxPduCanIdMask {CANIF_RXPDU_CANID_MASK} [ECUC_CanIf_00822]		
Description	Identifier mask which denotes relevant bits in the CAN Identifier. This parameter defines a CAN Identifier range in an alternative way to CanIfRxPduCanIdRange. It identifies the bits of the configured CAN Identifier that must match the received CAN Identifier. Range: 11 bits for Standard CAN Identifier, 29 bits for Extended CAN Identifier.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 536870911		
Default Value	536870911		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

Name	CanIfRxPduCanIdType {CANIF_RXPDU_CANIDTYPE} [ECUC_CanIf_00596]		
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	CAN frame with extended identifier (29 bits)	
	STANDARD_CAN	CAN frame with standard identifier (11 bits)	
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

Name	CanIfRxPduDlc {CANIF_RXPDU_DLC} [ECUC_CanIf_00599]		
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			
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

Name	CanIfRxPduHrhIdRef {CANIF_RXPDU_HRH_ID_REF} [ECUC_CanIf_00602]		
Description	The HRH to which Rx L-PDU belongs to, is referred through this parameter.		
Multiplicity	1		
Type	Reference to CanIfHrhCfg		
Configuration Class	**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: This information has to be derived from the CAN Driver configuration.		

Name	CanIfRxPduId {CANIF_RXPDUID} [ECUC_CanIf_00597]		
Description	ECU wide unique, symbolic handle for receive CAN L-SDU. 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			
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfRxPduReadData {CANIF_RXPDU_READDATA} [ECUC_CanIf_00600]		
Description	Enables and disables the Rx buffering for reading of received L-SDU data. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**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.		

Name	CanIfRxPduReadNotifyStatus {CANIF_RXPDU_READ_NOTIFYSTATUS} [ECUC_CanIf_00595]		
Description	Enables and disables receive indication for each receive CAN L-SDU for reading its notification status. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**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: CANIF_READRXPDU_NOTIFY_STATUS_API must be enabled.		

Name	CanIfRxPduRef {CANIF_RXPDU_REF} [ECUC_CanIf_00601]		
Description	Reference to the "global" Pdu structure to allow harmonization of handle IDs in the COM-Stack.		
Multiplicity	1		
Type	Reference to Pdu		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

Name	CanIfRxPduUserRxIndicationName {CANIF_RXPDU_USERRXINDICATION_NAME} [ECUC_CanIf_00530]		
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, J1939NM 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			
Length	1–32		
Regular Expression			
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfRxPduUserRxIndicationUL {CANIF_RXPDU_USERRXINDICATION_UL} [ECUC_CanIf_00529]	
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 Driver
	J1939NM	J1939Nm
	J1939TP	J1939Tp
	PDUR	PDU Router

	XCP	Extended Calibration Protocol	
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: ECU		

Included Containers		
Container Name	**Multiplicity**	**Scope / Depedency**
CanIfRxPduCanIdRange	0..1	Optional container that allows to map a range of CAN Ids to one PduId.
CanIfTTRxFrame Triggering	0..1	CanIfTTRxFrameTriggering is specified in the SWS TTCAN Interface and defines Frame trigger for TTCAN reception. This container is only included and valid if TTCAN is supported by the controller, enabled (see CanIfSupportTTCAN, ECUC_CanIf_00675), and a joblist is used for reception.

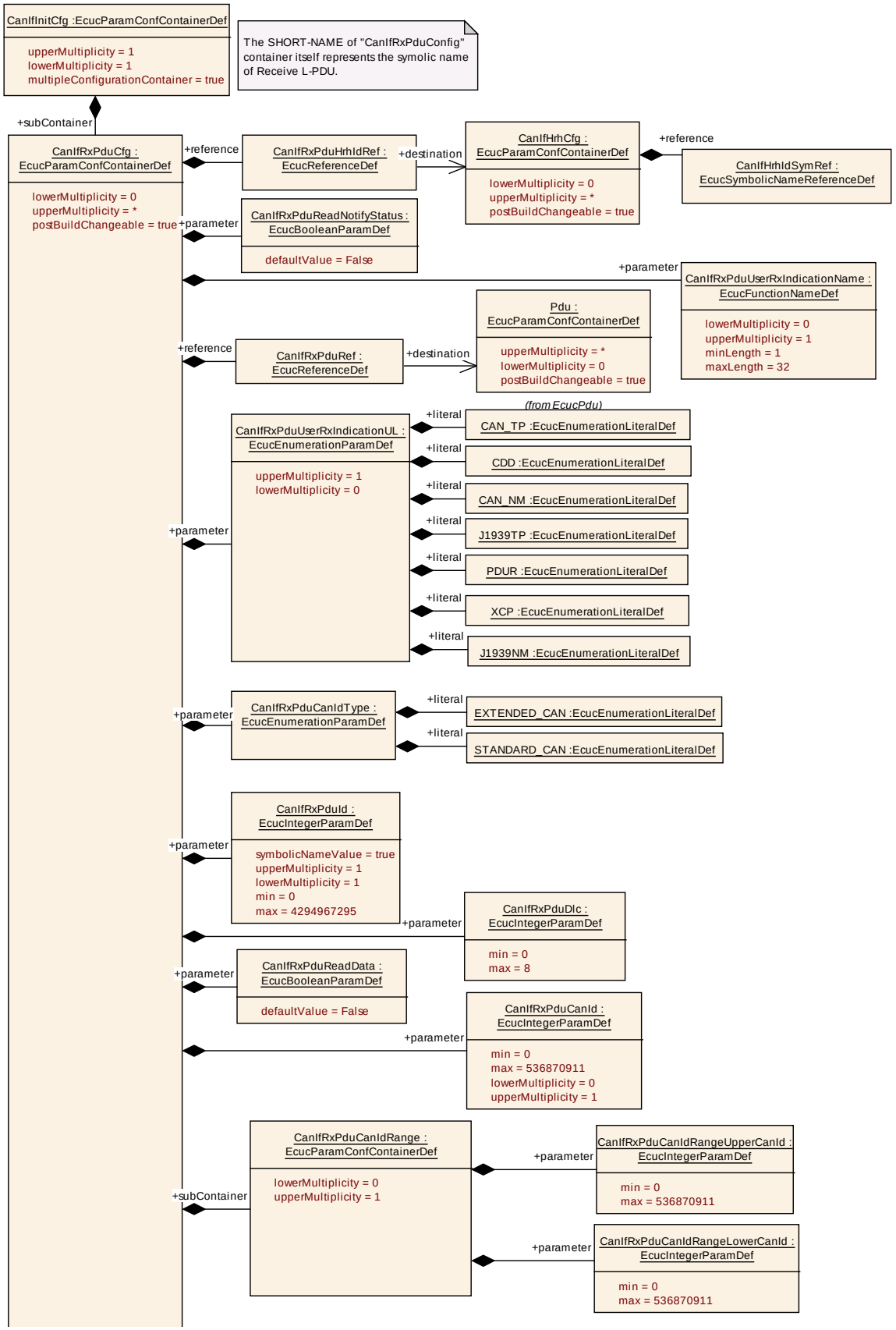

Figure 10.7: AR_EcucDef_CanIfRxPduCfg

CanIfRxPduCanIdRange

SWS Item	[ECUC_CanIf_00743]
Container Name	CanIfRxPduCanIdRange
Description	Optional container that allows to map a range of CAN Ids to one PduId.
Configuration Parameters	

Name	CanIfRxPduCanIdRangeLowerCanId {CANIF_RX_PDU_CANID_RANGE_LOWER_CANID} [ECUC_CanIf_00745]		
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			
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

Name	CanIfRxPduCanIdRangeUpperCanId {CANIF_RX_PDU_CANID_RANGE_UPPER_CANID} [ECUC_CanIf_00744]		
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			
Configuration Class	**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

CanIfDispatchCfg

SWS Item	[ECUC_CanIf_00250]
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	

Name	CanIfDispatchUserCheckTrcvWakeFlagIndicationName {CANIF_DISPATCH_USERCHECKTRCVWAKEFLAGINDICATION_NAME} [ECUC_CanIf_00791]		
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			
Regular Expression			
Configuration Class	**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_USERCHECKTRCVWAKEFLAGINDICATION_UL, CANIF_PUBLIC_PN_SUPPORT		

Name	CanIfDispatchUserCheckTrcvWakeFlagIndicationUL {CANIF_DISPATCH_USERCHECKTRCVWAKEFLAGINDICATION_UL} [ECUC_CanIf_00792]		
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 Driver	
Configuration Class	**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_PUBLIC_PN_SUPPORT		

Name	CanIfDispatchUserClearTrcvWufFlagIndicationName {CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_NAME} [ECUC_CanIf_00789]		
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			
Regular Expression			
Configuration Class	**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_USERCLEARTRCVWUFFLAGINDICATION_UL, CANIF_PUBLIC_PN_SUPPORT		

Name	CanIfDispatchUserClearTrcvWufFlagIndicationUL {CANIF_DISPATCH_USERCLEARTRCVWUFFLAGINDICATION_UL} [ECUC_CanIf_00790]		
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 Driver	
Configuration Class	**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_PUBLIC_PN_SUPPORT		

Name	CanIfDispatchUserConfirmPnAvailabilityName {CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_NAME} [ECUC_CanIf_00819]		
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			
Regular Expression			
Configuration Class	**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_USERCONFIRMPNAVAILABILITY_UL, CANIF_PUBLIC_PN_SUPPORT		

Name	CanIfDispatchUserConfirmPnAvailabilityUL {CANIF_DISPATCH_USERCONFIRMPNAVAILABILITY_UL} [ECUC_CanIf_00820]		
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 Driver	
Configuration Class	**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_PUBLIC_PN_SUPPORT		

Name	CanIfDispatchUserCtrlBusOffName {CANIF_DISPATCH_USERCTRLBUSOFF_NAME} [ECUC_CanIf_00525]		
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			
Length	1–32		
Regular Expression			

Configuration Class	**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		

Name	CanIfDispatchUserCtrlBusOffUL {CANIF_DISPATCH_USERCTRLBUSOFF_UL} [ECUC_CanIf_00547]		
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 Driver	
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfDispatchUserCtrlModeIndicationName {CANIF_DISPATCH_USERCTRLMODEINDICATION_NAME} [ECUC_CanIf_00683]		
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			
Length	1–32		
Regular Expression			
Configuration Class	**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		

Name	CanIfDispatchUserCtrlModeIndicationUL {CANIF_DISPATCH_USERCTRLMODEINDICATION_UL} [ECUC_CanIf_00684]		
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 Driver	
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfDispatchUserTrcvModeIndicationName {CANIF_DISPATCH_USERTRCVMODEINDICATION_NAME} [ECUC_CanIf_00685]		
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			
Length	1–32		
Regular Expression			
Configuration Class	**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		

Name	CanIfDispatchUserTrcvModeIndicationUL {CANIF_DISPATCH_USERTRCVMODEINDICATION_UL} [ECUC_CanIf_00686]	
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 Driver	
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfDispatchUserValidateWakeupEventName {CANIF_DISPATCH_USERVALIDATEWAKEUPEVENT_NAME} [ECUC_CanIf_00531]		
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			
Length	1–32		
Regular Expression			
Configuration Class	**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		

Name	CanIfDispatchUserValidateWakeupEventUL {CANIF_DISPATCH_USERVALIDATEWAKEUPEVENT_UL} [ECUC_CanIf_00549]		
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 Driver	
	ECUM	ECU State Manager	
Configuration Class	**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

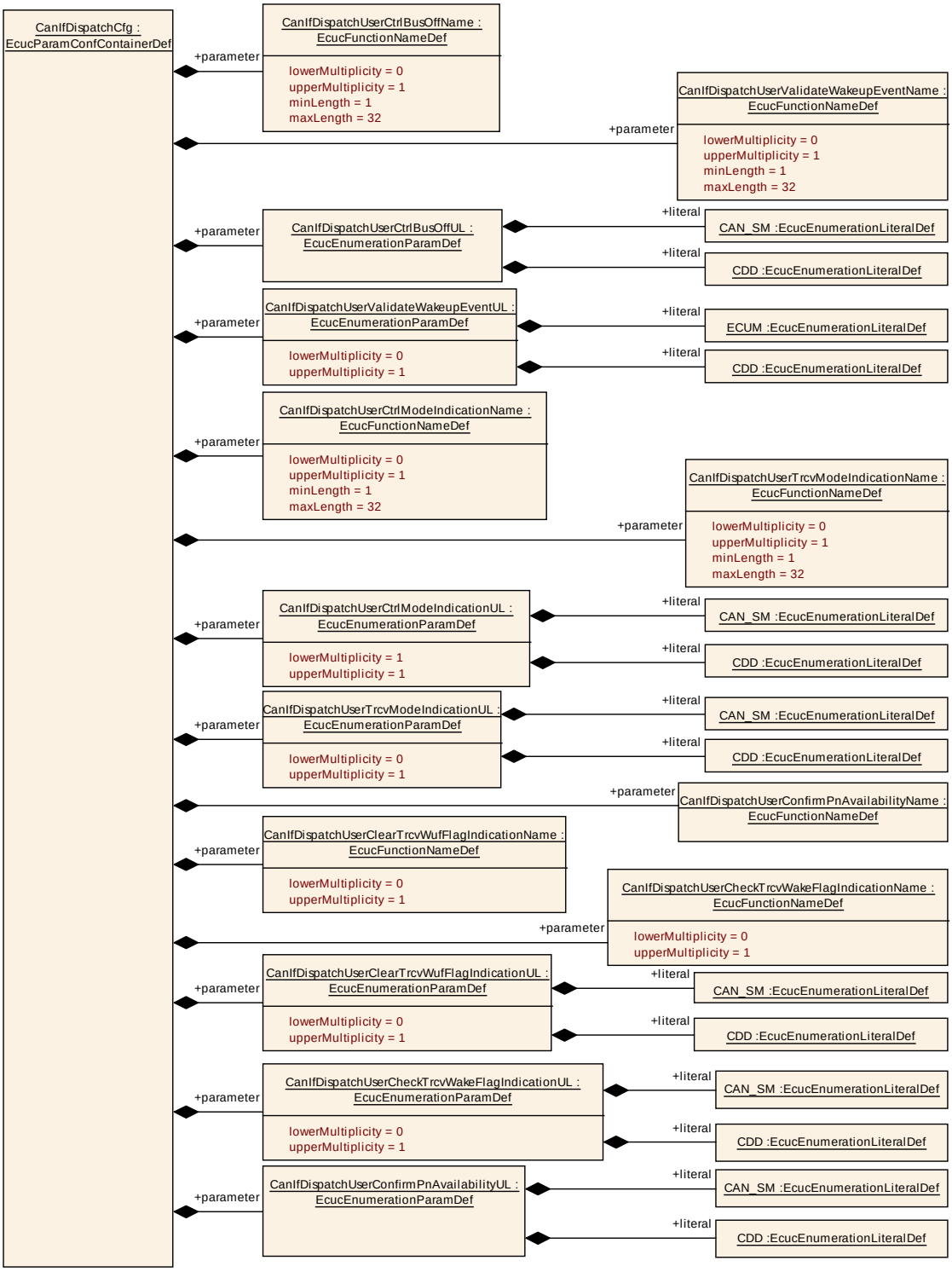

Figure 10.8: AR_EcucDef_CanIfDispatchCfg

CanIfCtrlCfg

SWS Item	[ECUC_CanIf_00546]
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. **Attributes:** postBuildChangeable=false
Configuration Parameters	

Name	CanIfCtrlCanCtrlRef {CANIF_CTRL_CAN_CONTROLLER_REF} [ECUC_CanIf_00636]		
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	Symbolic name reference to CanController		
Configuration Class	**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		

Name	CanIfCtrlId {CANIF_CTRL_ID} [ECUC_CanIf_00647]		
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 .. 255		
Default Value			
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfCtrlWakeupSupport {CANIF_CTRL_WAKEUP_SUPPORT} [ECUC_CanIf_00637]		
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		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: ECU		

No Included Containers

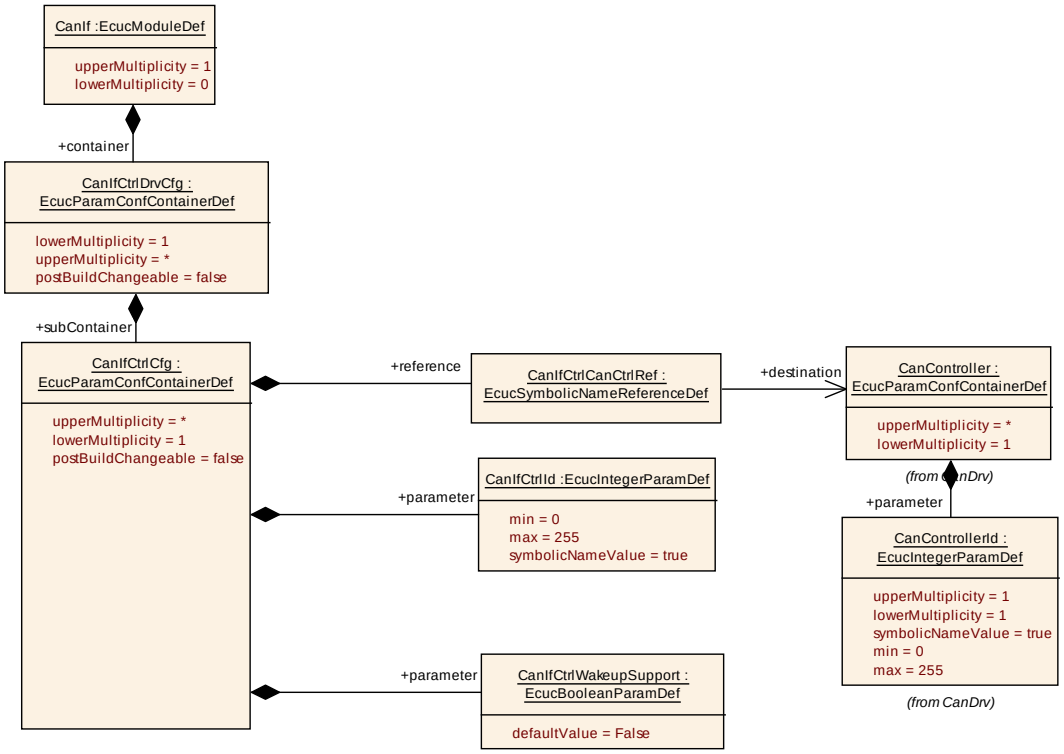

Figure 10.9: AR_EcucDef_CanIfCtrlCfg

CanIfCtrlDrvCfg

SWS Item	[ECUC_CanIf_00253]
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. **Attributes:** postBuildChangeable=false
Configuration Parameters	

Name	CanIfCtrlDrvInitHohConfigRef {CANIF_CTRLDRV_INIT_HOH_CONFIG_REF} [ECUC_CanIf_00642]		
Description	Reference to the Init Hoh Configuration		
Multiplicity	1		
Type	Reference to CanIfInitHohCfg		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: local		

Name	CanIfCtrlDrvNameRef {CANIF_CTRLDRV_NAME_REF} [ECUC_CanIf_00638]		
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		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local		

Name	CanIfCtrlDrvTxCancellation {CANIF_CTRLDRV_TX_CANCELLATION} [ECUC_CanIf_00640]
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	

Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local dependency: CANIF_PUBLIC_TX_BUFFERING has to be enabled		

Included Containers		
Container Name	**Multiplicity**	**Scope / Depedency**
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.

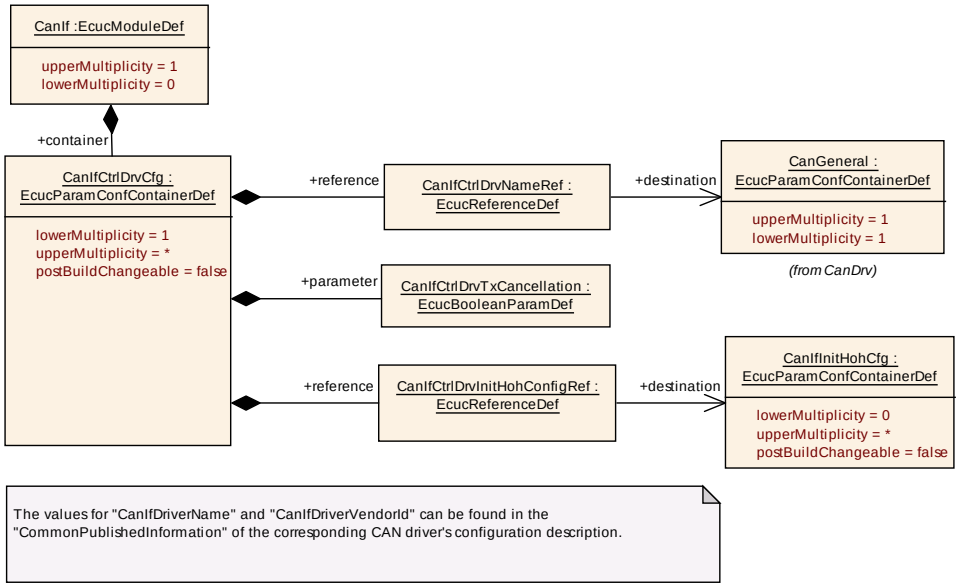

Figure 10.10: AR_EcucDef_CanIfCtrlDrvCfg

CanIfTrcvDrvCfg

SWS Item	[ECUC_CanIf_00273]
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. **Attributes:** postBuildChangeable=false
Configuration Parameters	

Included Containers		
Container Name	**Multiplicity**	**Scope / Depedency**

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.

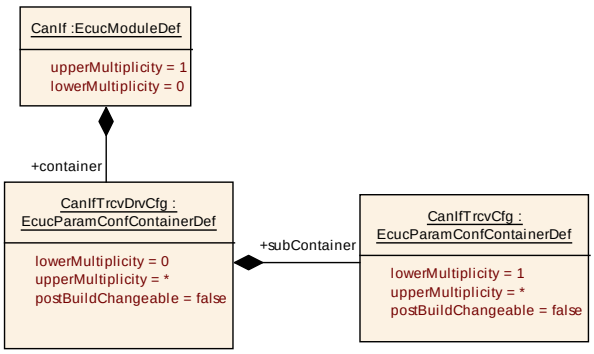

Figure 10.11: AR_EcucDef_CanIfTrcvDrvCfg

CanIfTrcvCfg

SWS Item	[ECUC_CanIf_00587]
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. **Attributes:** postBuildChangeable=false
Configuration Parameters	

Name	CanIfTrcvCanTrcvRef {CANIF_TRCV_CAN_TRANSCEIVER_REF} [ECUC_CanIf_00605]		
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	Symbolic name reference to CanTrcvChannel		
Configuration Class	**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		

Name	CanIfTrcvId {CANIF_TRCV_ID} [ECUC_CanIf_00654]		
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 .. 255		
Default Value			
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfTrcvWakeupSupport {CANIF_TRCV_WAKEUP_SUPPORT} [ECUC_CanIf_00606]		
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		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: ECU		

No Included Containers

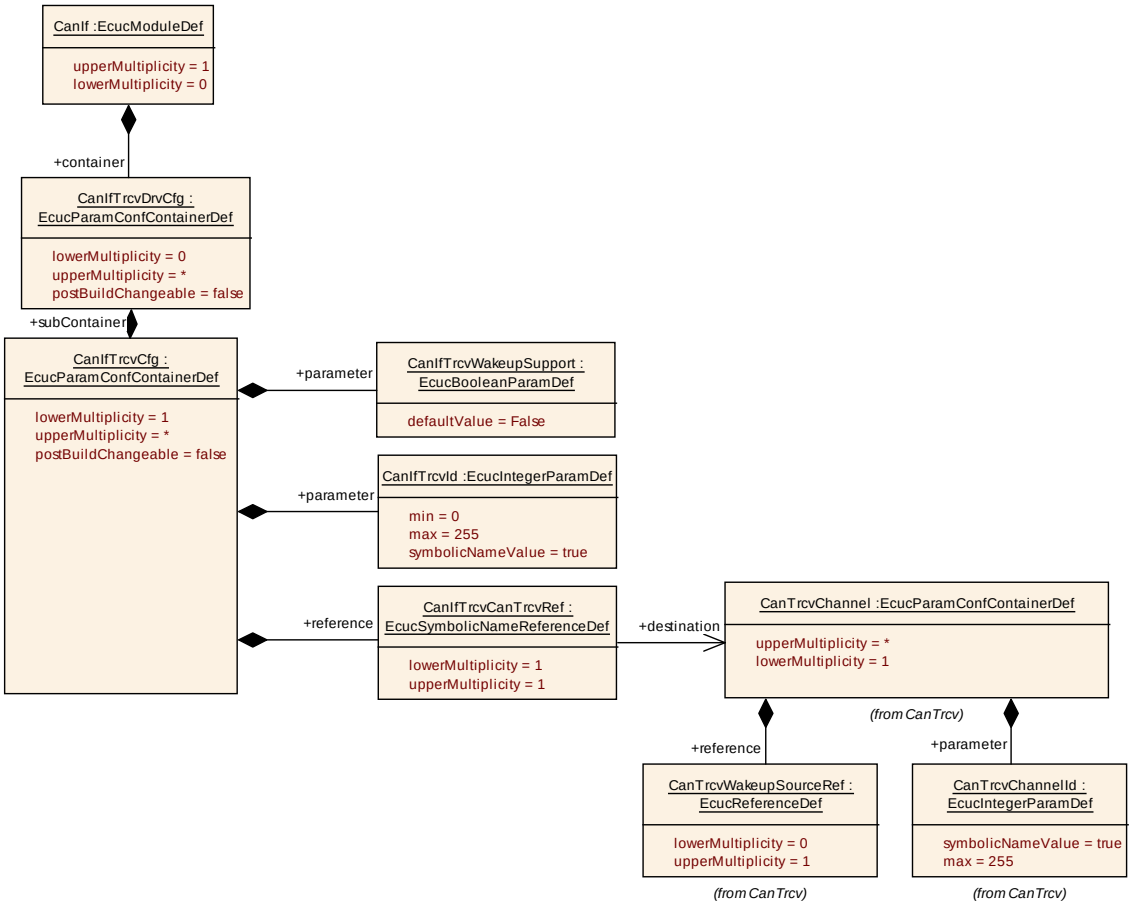

Figure 10.12: AR_EcucDef_CanIfTrcvCfg

CanIfInitHohCfg

SWS Item	[ECUC_CanIf_00257]
Container Name	CanIfInitHohCfg {CANIF_INIT_HOH_CFG}
Description	This container contains the references to the configuration setup of each underlying CAN Driver. **Attributes:** postBuildChangeable=false
Configuration Parameters	

Name	CanIfInitRefCfgSet {CANIF_INIT_REF_CFGSET} [ECUC_CanIf_00620] (Obsolete)		
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. **Tags:** atp.Status=obsolete atp.StatusComment=This parameter is set to obsolete and will be removed in the next major release. atp.StatusRevisionBegin=4.1.2		
Multiplicity	1		
Type	Reference to CanConfigSet		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

Included Containers		
Container Name	**Multiplicity**	**Scope / Depedency**
CanIfHrhCfg	0..*	This container contains configuration parameters for each hardware receive object (HRH).
CanIfHthCfg	0..*	This container contains parameters related to each HTH.

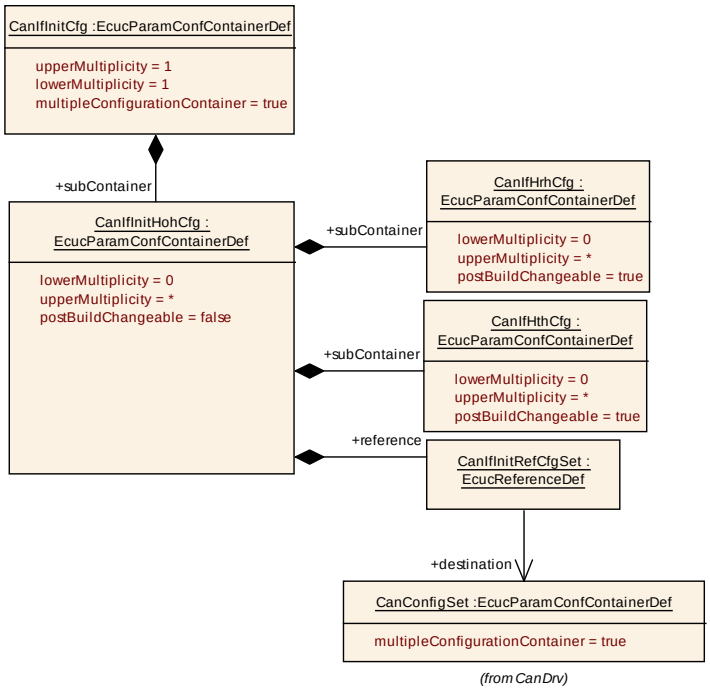

Figure 10.13: AR_EcucDef_CanIfInitHohCfg

CanIfHthCfg

SWS Item	[ECUC_CanIf_00258]
Container Name	CanIfHthCfg {CanInterfaceHthConfiguration}
Description	This container contains parameters related to each HTH. **Attributes:** postBuildChangeable=true
Configuration Parameters	

Name	CanIfHthCanCtrlIdRef {CANIF_HTH_CAN_CONTROLLER_ID_REF} [ECUC_CanIf_00625]		
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		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfHthCanHandleTypeRef {CANIF_HTH_HANDLETYPE_REF} [ECUC_CanIf_00626] (Obsolete. Use CanIfHthIdSymRef instead.)		
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. **Tags:** atp.Status=obsolete atp.StatusComment=CanIfHthIdSymRef shall be used instead to get the CanHandleType and CanObjectId of CAN Driver. atp.StatusRevisionBegin=4.0.3		
Multiplicity	0..1		
Type	Symbolic name reference to CanHardwareObject		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: local		

Name	CanIfHthIdSymRef {CANIF_HTH_ID_SYMREF} [ECUC_CanIf_00627]		
Description	The parameter refers to a particular HTH object in the CanDrv configuration (see CanHardwareObject ECUC_Can_00324). `CanIf receives the following information of the CanDrv module by this reference:` • CanHandleType (see ECUC_Can_00323) • CanObjectId (see ECUC_Can_00326)		
Multiplicity	1		
Type	Symbolic name reference to CanHardwareObject		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: ECU		

No Included Containers

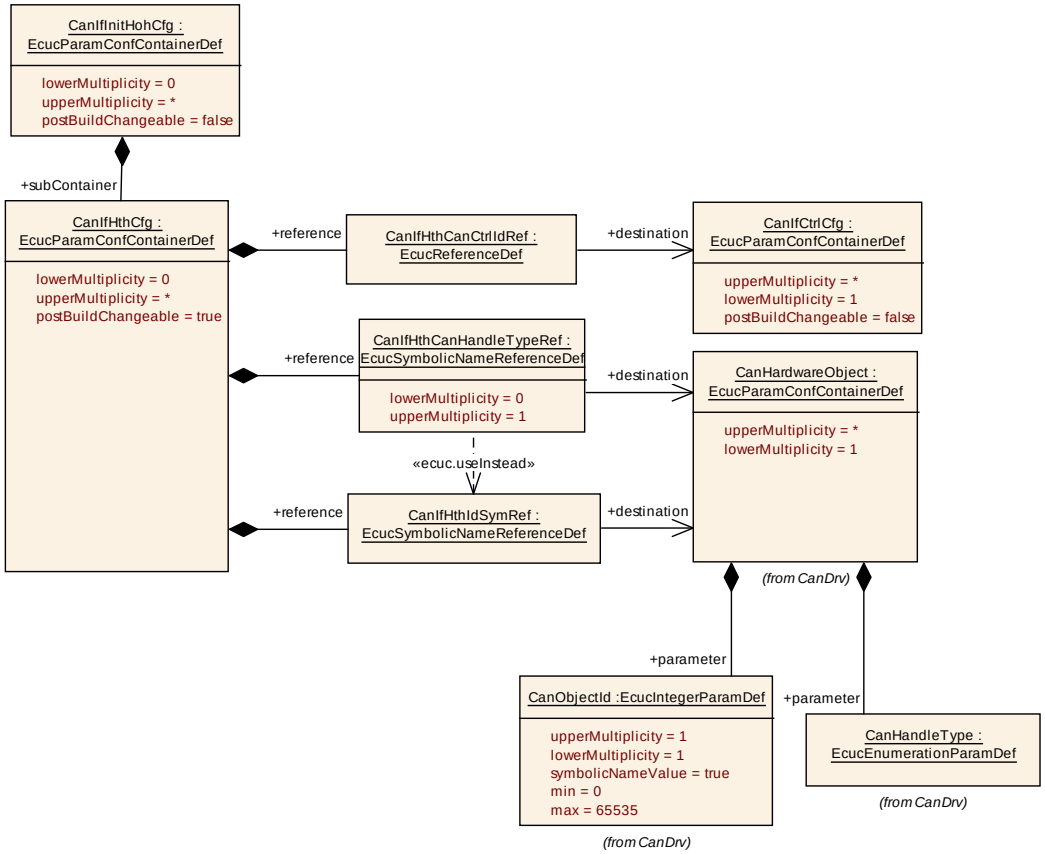

Figure 10.14: AR_EcucDef_CanIfHthCfg

CanIfHrhCfg

SWS Item	[ECUC_CanIf_00259]
Container Name	CanIfHrhCfg {CanInterfaceHrhConfiguration}
Description	This container contains configuration parameters for each hardware receive object (HRH). **Attributes:** postBuildChangeable=true
Configuration Parameters	

Name	CanIfHrhCanCtrlIdRef {CANIF_HRH_CAN_CTRL_ID_REF} [ECUC_CanIf_00631]		
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		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	CanIfHrhCanHandleTypeRef {CANIF_HRH_HANDLETYPE_REF} [ECUC_CanIf_00633] (Obsolete. Use CanIfHrhIdSymRef instead.)		
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. CanIfHrhIdSymRef shall be used instead to get the CanHandleType and CanObjectId of CAN Driver. In the next major release this reference will be deleted. **Tags:** atp.Status=obsolete atp.StatusComment=CanIfHrhIdSymRef shall be used instead to get the CanHandleType and CanObjectId of CAN Driver. atp.StatusRevisionBegin=4.0.3		
Multiplicity	0..1		
Type	Symbolic name reference to CanHardwareObject		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: local		

Name	CanIfHrhIdSymRef {CANIF_HRH_ID_SYMREF} [ECUC_CanIf_00634]		
Description	The parameter refers to a particular HRH object in the CanDrv configuration (see CanHardwareObject ECUC_Can_00324). `CanIf receives the following information of the CanDrv module by this reference:` • CanHandleType (see ECUC_Can_00323) • CanObjectId (see ECUC_Can_00326)		
Multiplicity	1		
Type	Symbolic name reference to CanHardwareObject		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

Name	CanIfHrhSoftwareFilter {CANIF_HRH_SOFTWARE_FILTER} [ECUC_CanIf_00632]		
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		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	–	
Scope / Dependency	scope: local		

Included Containers		
Container Name	**Multiplicity**	**Scope / Depedency**
CanIfHrhRangeCfg	0..*	Defines the parameters required for configurating multiple CANID ranges for a given same HRH.

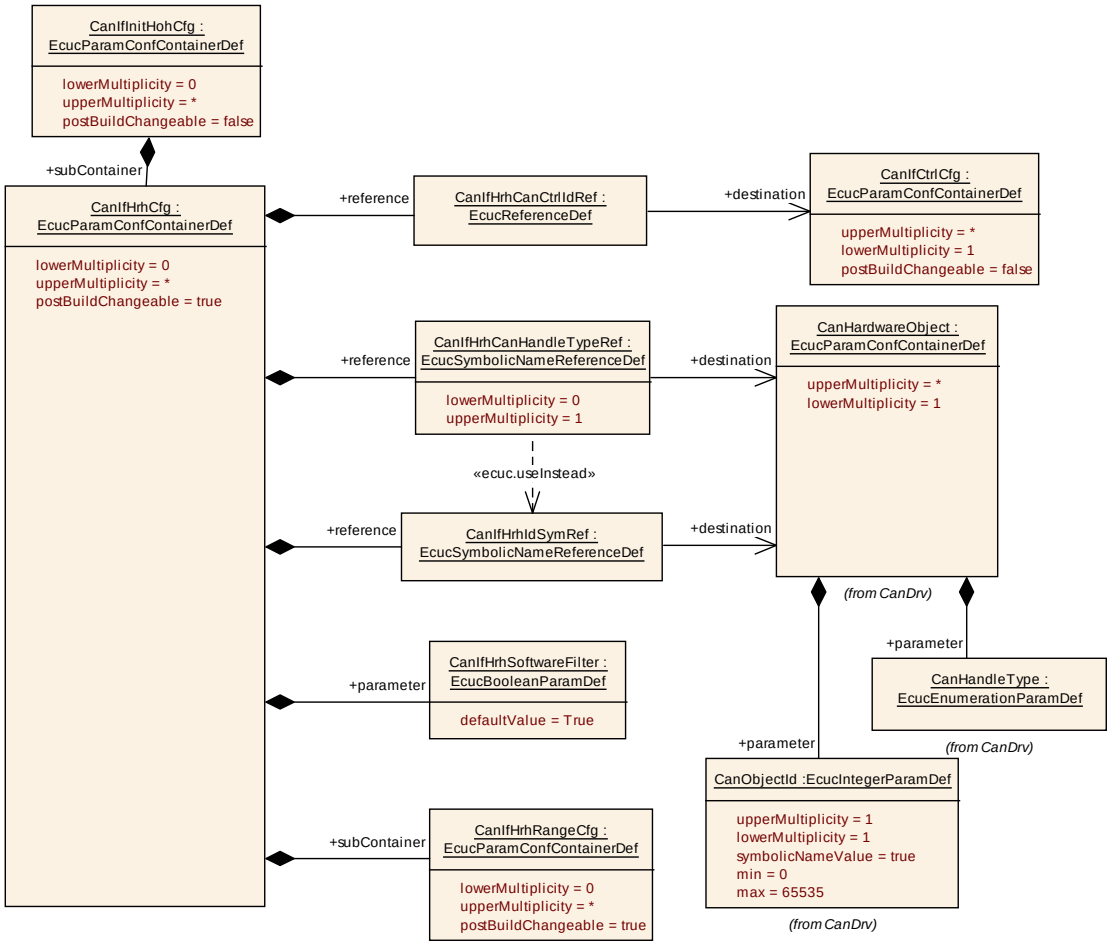

Figure 10.15: AR_EcucDef_CanIfHrhCfg

CanIfHrhRangeCfg

SWS Item	[ECUC_CanIf_00628]
Container Name	CanIfHrhRangeCfg {CanInterfaceHrhRangeConfiguration}
Description	Defines the parameters required for configurating multiple CANID ranges for a given same HRH. **Attributes:** postBuildChangeable=true
Configuration Parameters	

Name	CanIfHrhRangeBaseId {CANIF_HRHRANGE_BASEID} [ECUC_CanIf_00825]	
Description	CAN Identifier used as base value in combination with CanIfHrhRangeMask for a masked ID range in which all CAN Ids shall pass the software filtering. The size of this parameter is limited by CanIfHrhRangeRxPduRangeCanIdType.	
Multiplicity	0..1	
Type	EcucIntegerParamDef	
Range	0 .. 536870911	

Default Value			
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

Name	CanIfHrhRangeMask {CANIF_HRHRANGE_MASK} [ECUC_CanIf_00826]		
Description	Used as mask value in combination with CanIfHrhRangeBaseId for a masked ID range in which all CAN Ids shall pass the software filtering. The size of this parameter is limited by CanIfHrhRangeRxPduRangeCanIdType.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 536870911		
Default Value			
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

Name	CanIfHrhRangeRxPduLowerCanId {CANIF_HRHRANGE_LOWER_CANID} [ECUC_CanIf_00629]		
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	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 536870911		
Default Value			
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

Name	CanIfHrhRangeRxPduRangeCanIdType {CANIF_HRHRANGE_CANIDTYPE} [ECUC_CanIf_00644]		
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).	
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

Name	CanIfHrhRangeRxPduUpperCanId {CANIF_HRHRANGE_UPPER_CANID} [ECUC_CanIf_00630]		
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	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 536870911		
Default Value			
Configuration Class	**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

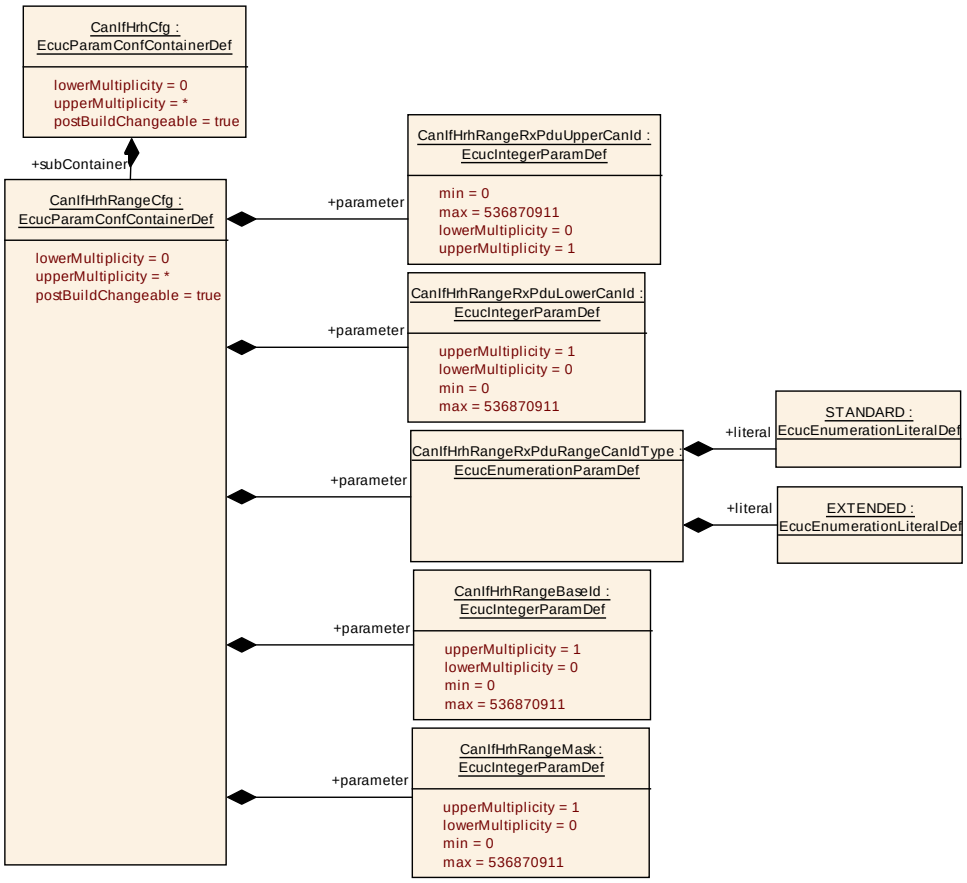

Figure 10.16: AR_EcucDef_CanIfHrhRangeCfg

CanIfBufferCfg

SWS Item	[ECUC_CanIf_00832]
Container Name	CanIfBufferCfg {CANIF_BUFFER_CFG}
Description	This container contains the Txbuffer configuration. Multiple buffers with different sizes could be configured. If CanIfBufferSize (ECUC_CanIf_00834) equals 0, the CanIf Tx L-PDU only refers via this CanIfBufferCfg the corresponding CanIfHthCfg. **Attributes:** postBuildChangeable=true
Configuration Parameters	

Name	CanIfBufferHthRef {CANIF_BUFFER_HTH_REF} [ECUC_CanIf_00833]		
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		
Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

Name	CanIfBufferSize {CANIF_BUFFER_SIZE} [ECUC_CanIf_00834]		
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		
Configuration Class	**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		

No Included Containers

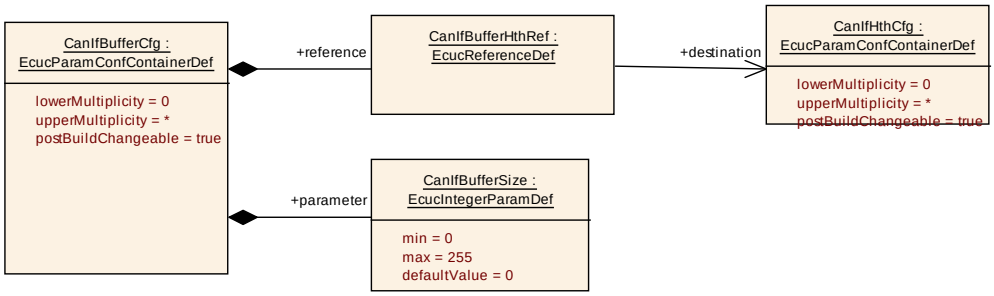

Figure 10.17: AR_EcucDef_CanIfBufferCfg

A Not applicable requirements

[SWS_CANIF_00999] ⌈ These requirements are not applicable to this specification. ⌋*(SRS_BSW_00159, SRS_BSW_00167, SRS_BSW_00170, SRS_BSW_00416, SRS_BSW_00168, SRS_BSW_00423, SRS_BSW_00424, SRS_BSW_00425, SRS_BSW_00426, SRS_BSW_00427, SRS_BSW_00428, SRS_BSW_00429, BSW00431, SRS_BSW_00432, SRS_BSW_00433, BSW00434, SRS_BSW_00336, SRS_BSW_00417, SRS_BSW_00164, SRS_BSW_00326, SRS_BSW_00007, SRS_BSW_00307, SRS_BSW_00373, SRS_BSW_00435, SRS_BSW_00328, SRS_BSW_00378, SRS_BSW_00306, SRS_BSW_00308, SRS_BSW_00309, SRS_BSW_00376, SRS_BSW_00330, SRS_BSW_00172, SRS_BSW_00010, SRS_BSW_00341, SRS_BSW_00334, SRS_CAN_01139, SRS_CAN_01014, BSW01024)*