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

Document Version	3.0.3
Document Status	Final
Part of Release	3.1
Revision	0001

Document Change History

Date	Version	Changed by	Change Description
04.08.2008	3.0.3	AUTOSAR Administration	Layout adaptations
23.06.2008	3.0.2	AUTOSAR Administration	Legal disclaimer revised
29.01.2008	3.0.1	AUTOSAR Administration	• Replaced chapter 10 content with generated tables from AUTOSAR MetaModel.
12.12.2007	3.0.0	AUTOSAR Administration	• Interface abstraction: network related interface changed into a controller related one • Wakeup mechanism completely reworked, APIs added & changed for Wakeup • Initialization changed (flat initialization) • Scheduled main functions skipped due to changed BSW Scheduler responsibility • Document meta information extended • Small layout adaptations made

Document Change History			
Date	**Version**	**Changed by**	**Change Description**
31.10.2007	2.1.0	AUTOSAR Administration	• Header file structure changed • Support of mixed mode operation (StandardCAN & Extended CAN in parallel on one network) added • Support of CAN Transceiver according AUTOSAR_WP1.1.2_SoftwareArchitecture.ppt • API <User>_DlcErrorNotification deleted • Pre-compile/Link-Time/Post-Built definiton 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

Page left intentionally blank

Disclaimer

Advice to users of AUTOSAR Specification Documents:

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

1 Introduction and functional overview

CANIF143: This specification describes the functionality, API and the configuration for the AUTOSAR Basic Software module CAN Interface.

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

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

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

The CAN Interface 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, so that underlying CAN device drivers only focus on access and control of the corresponding specific CAN hardware device.

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

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

The CAN Interface 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 to e.g. the corresponding NM module.

2 Acronyms and Abbreviations

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

Acronym:	***Description:***
Buffering	Buffer for a single data unit, for example CAN ID, DLC and SDU, is stored at a dedicated memory address in RAM.
CAN communication matrix	Describes the complete CAN network: ▪ Participating nodes ▪ Definition of all CAN PDUs (identifier, DLC) ▪ Source and Sinks for PDUs
CAN controller	A CAN controller is a CPU on-chip or external standalone hardware device. One CAN controller is connected to one physical channel.
CAN device driver	Generic term of CAN Driver and CAN Transceiver Driver.
CAN hardware unit	A CAN Hardware unit may consist of one or multiple CAN controllers of the same type and one, two or multiple CAN RAM areas. The CAN hardware unit is located on-chip or as external device. The CAN hardware unit is represented by one CAN Driver.
CAN L-PDU	CAN Protocol Data Unit. Consists of an identifier, DLC and data (SDU).
CAN L-SDU	CAN Service Data Unit. Data that are transported inside the CAN L-PDU.
FIFO	First-In-First-Out
Hardware 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 hardware objects that are configured as 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.
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.

Abbreviation:	***Description:***
BSW	Basic Software
CANIF	CAN Interface
DLC	Data Length Code (part of CAN L-PDU that describes the SDU length)
DLL	Data Link Layer
HOH	CAN hardware object handle
HRH	CAN hardware receive handle

HTH	CAN hardware transmit handle
ISR	Interrupt service routine
L-PDU	Protocol Data Unit for the data link layer (DLL)
L-SDU	Service Data Unit for the data link layer (DLL)
PDU	Protocol Data Unit
SDU	Service Data Unit

3 Related documentation

3.1 Input documents

[1] List of Basic Software Modules
AUTOSAR_BasicSoftwareModules.pdf

[2] Layered Software Architecture
AUTOSAR_LayeredSoftwareArchitecture.pdf

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

[4] Specification of Standard Types
AUTOSAR_SWS_StandardTypes.pdf

[5] Specification of Communication Stack Types
AUTOSAR_SWS_ComStackTypes.pdf

[6] Specification of ECU Configuration
AUTOSAR_ECU_Configuration.pdf

[7] Requirements on CAN
AUTOSAR_SRS_CAN.pdf

[8] Specification of CAN Driver
AUTOSAR_SWS_CAN_Driver.pdf

[9] Specification of CAN Transceiver Driver
AUTOSAR_SWS_CAN_TransceiverDriver.pdf

[10] Specification of CAN Transport Layer
AUTOSAR_SWS_CAN_TP.pdf

[11] Specification of CAN State Manager
AUTOSAR_SWS_CAN_StateManager.pdf

[12] Specification of CAN Network Management
AUTOSAR_SWS_CAN_NM.pdf

[13] Specification of Generic Network Management
AUTOSAR_SWS_Generic_NM.pdf

[14] Specification of Communication
AUTOSAR_SWS_COM.pdf

[15] Specification of ECU State Manager
AUTOSAR_SWS_ECU_StateManager.pdf

[16] Specification of BSW Scheduler

AUTOSAR_SWS_BSW_Scheduler.pdf

[17] AUTOSAR Basic Software Module Description Template, AUTOSAR_BSW_Module_Description.pdf

3.2 Related standards and norms

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

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

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

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

4 Constraints and assumptions

4.1 Limitations

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

4.2 Applicability to car domains

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

5 Dependencies to other modules

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

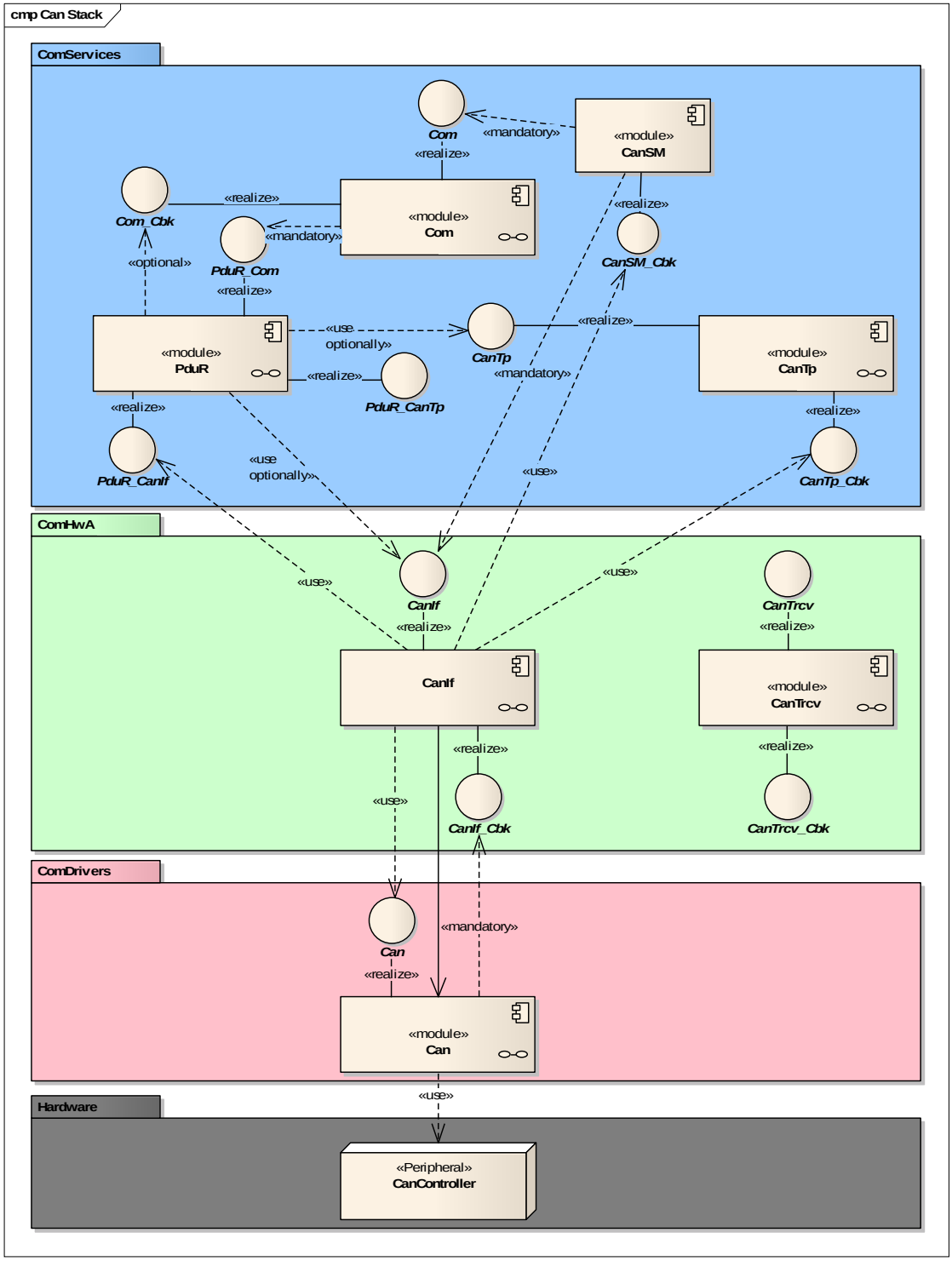

Figure 2 CANIF dependencies in AUTOSAR BSW

5.1 Upper Protocol Layers

CANIF193: Inside the AUTOSAR BSW architecture the upper layers of the CAN Interface are represented by the PDU Router, CanNm, CanTp, CanSM and EcuM.

CANIF037: The AUTOSAR BSW architecture indicates, that the application data buffers are located in the upper layer which they belong to. Direct access to these buffers is prohibited. The buffer location is passed by the CAN Interface from or to the CAN Driver during transmission and reception. During execution of these transmission/reception indication services buffer location is passed. Data integrity is guaranteed by used of lock mechanisms each time the buffer has been accessed. See [7.18 Data integrity].

CANIF192: The API used by the CAN Interface consists of notification services as basic agents for the transfer of CAN related data (i.e. CAN ID, CAN DLC) to the target upper layer. The call parameters of these services points to the information buffered in the CAN Driver or they refer directly to the CAN hardware.

5.2 Initialization: Ecu State Manager

CANIF283: The EcuM initializes the CanIf (refer to [15] Specification of ECU State Manager).

5.3 Mode Control: CAN State Manager

CANIF299: The CanSM is responsible for mode control management of all supported CAN controllers, for startup, wakeup and as well for sleep transitions.

5.4 Lower layers: CAN Driver

CANIF034: The main lower layer CAN device driver is represented by the CAN Driver (see [8] Specification of CAN Driver). The CAN Interface has a close relation to the CAN Driver as a result of its position in the AUTOSAR Basic Software Architecture. Events detected and processed by the CAN Driver are forwarded to the CAN Interface.

CANIF267: The CAN Interface passes operation mode requests of the CanSM to the corresponding underlying CAN controllers. The CAN Driver provides a hardware abstracted access to the CAN controller only, but control of operation modes is done only in CanSM, neither in CAN Driver nor in CANIF.

CANIF191: The CAN Driver provides a normalized L-SDU to ensure hardware independence of the CAN Interface. The pointer to this normalized L-SDU points either to a temporary buffer (for e.g. data normalizing) or the CAN hardware

dependent to the CAN Driver. For the CAN Interface the kind of L-SDU buffer is invisible.

CANIF038: The CAN Interface provides notification services used by the CAN Driver in all notifications scenarios, for example: transmit confirmation, receive indication, BusOff notification, wakeup notification and wakeup validation notification.

CANIF106: In case of using multiple CAN Drivers serving different interrupt vectors these call-out services mentioned above must be re-entrant, refer to [7.25 Multiple CAN Driver support].

CANIF133: The call-out services called by the CAN Driver are declared and implemented inside the CAN Interface. The call-out services called by the CAN Interface are declared and placed inside the appropriate upper communication service layer, for example PduR, CanNm, CanTp.

CANIF271: 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 CAN Transceiver Driver is sufficient, whereas dependent to the type of the CAN controller devices one or two different CAN Drivers are needed (see 7.5 Physical channel view).

5.5 Lower layers: CAN Transceiver Driver

CANIF266: The second available lower layer CAN device driver is represented by the CAN Transceiver Driver (see [9] Specification of CAN Transceiver Driver).

CANIF268: Operation mode control of the CAN transceiver device is done by each CAN transceiver driver itself. The CAN Interface just maps all APIs of several underlying CAN Transceiver Drivers 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 CAN Transceiver Driver is done inside the CAN Interface.

CANIF277: The CAN Interface Layer maps the following services of all underlying CAN Transceiver Drivers to one unique interface. These are further described in the CAN Transceiver Driver SWS (see [9]Specification of CAN Transceiver Driver):

- Unique CAN Transceiver Driver mode request and read services to manage the operation modes of each underlying CAN transceiver device.
- Read service for CAN transceiver wakeup reason support.
- Mode request service to enable/disable/clear wakeup event state of each used CAN transceiver.

CANIF270: The call-out services for wakeup event notification is called by the CAN Transceiver Drivers and mapped by CAN Interface to the corresponding wakeup event notification implemented e.g. in the ECU State Manager.

5.6 Configuration

CANIF035: The CAN Interface design is optimized to manage CAN protocol specific capabilities and handling of the used underlying CAN controller.
The following standardized information is therefore retrieved from the CAN Driver configuration:

- 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.
- 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.
- Range of received CAN IDs passing hardware acceptance filter for each hardware object. The CAN Interface uses fixed assignments between HRHs and L-PDUs to be received in the corresponding hardware object to conduct a search algorithm (see 7.21 Software receive filter)

CANIF190: The CAN Interface Layer needs information about all used upper communication service layers and L-PDUs to be dispatched. The following information has to be set up at configuration time for integration of the CAN Interface module inside the AUTOSAR COM stack:

- Transmitting upper layer module and transmit I-PDU for each transmit L-PDU.
  => Used for dispatching of transmit confirmation services.
- Receiving upper layer module and and receive I-PDU for each receive L-PDU.
  => Used for L-PDU dispatching during receive indication.

CANIF036: The CAN Interface Layer 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:

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

5.7 File structure

5.7.1 Code file structure

CANIF151: The code file structure shall not be defined within this specification completely. Here it shall be pointed out that the code-file structure shall include the following files named:

- `CanIf_<X>.c` – for implementation of the provided functionality. The extensions <X> is optional for usage of multiple C-files.
- `CanIf_Cfg.c` – for pre-compile time configurable parameters and
- `CanIf_Lcfg.c` – for link time configurable parameters.
- `CanIf_Pbcfg.c` – for post build time configurable parameters.

All of these files shall contain all link time and post-build time configurable parameters.

CANIF152: The API of all used underlying CAN Drivers must be known latest at link-time.
The location of the API is provided for pre-compile time configuration either by using of external declaration in includes of all CAN Drivers public header files `can_<x>.h` or by the `CanIf_Cfg.c`.
The location of the API is provided for link time configuration by a set of function pointers for each CAN Driver. The values for these pointers are given at link time.

CANIF149: Constants and functions used internally by the CAN Interface without connection to the configuration files are declared in CanIf source file(s).

CANIF117: The include file structure can be constructed as shown in [CANIF141:].

5.7.2 Header file structure

CANIF116: The CAN Interface shall offer a header file `CanIf.h`, which includes all types and interfaces. This header file only contains extern declarations of constants, global data, type definitions and services that are specified in chapter [8 API specification].

CANIF141:

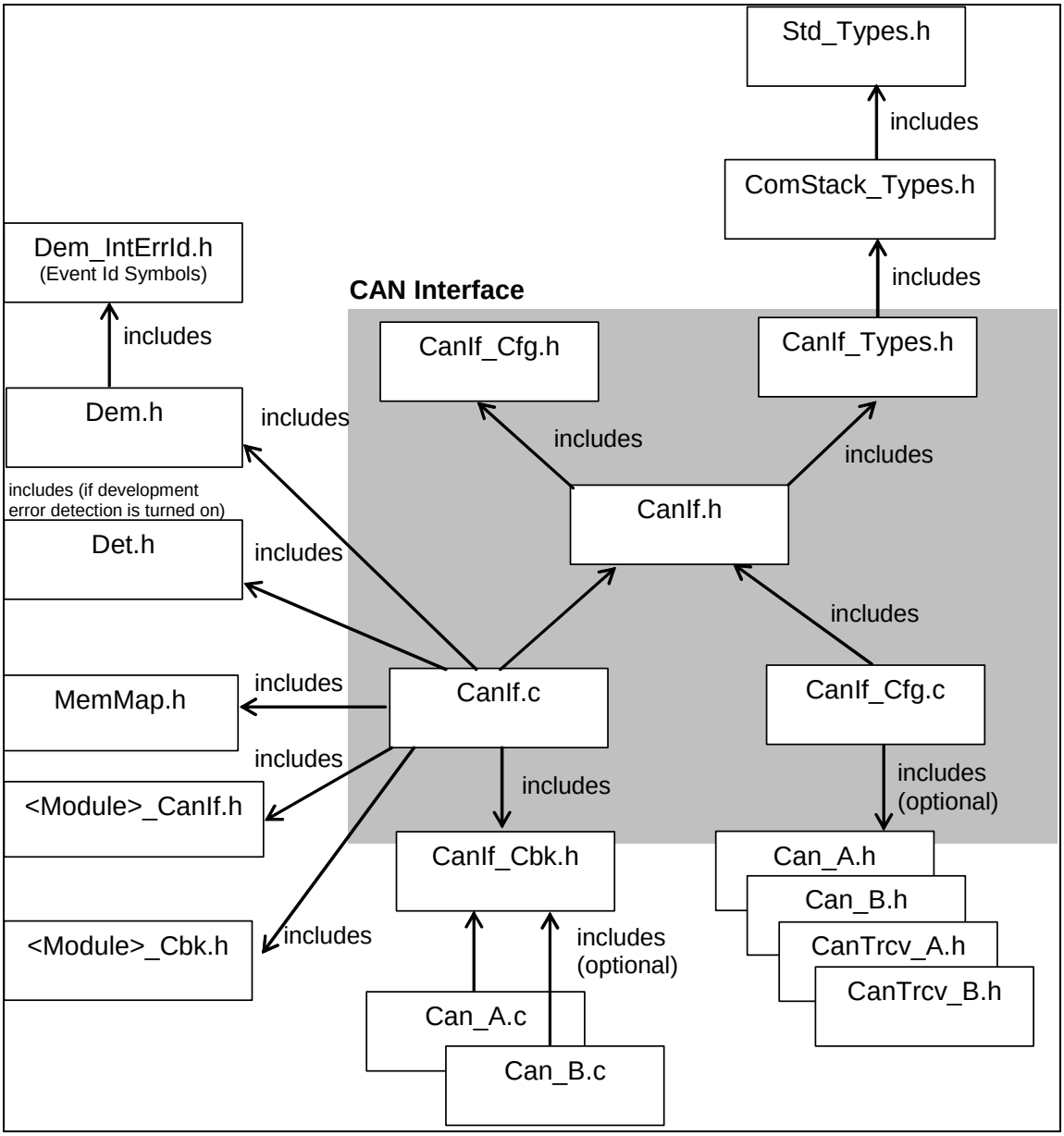

Figure 3 Code and include file structure

CANIF121: The CAN Interface shall offer a header file `CanIf_Cbk.h`, which declares the call-out functions called by the CAN Driver.

CANIF122: The CAN Interface includes all necessary configuration data by the header files

- `CanIf.h` – for declaration of the provided interface functions
- `CanIf_Cbk.h` – for declaration of the provided callout functions
- `CanIf_Cfg.h` – for pre-compile time configurable parameters and
- `CanIf_Lcfg.h` – for link build time configurable parameters
- `CanIf_Pbcfg.h` – for post build time configurable parameters

CANIF279: The CAN Interface shall include all headers files <Module.h> of those modules, from which declarations of API services or type definitions are needed:

- Can_A.h/Can_B.h – for services and type definitions of the CAN Driver
- CanTrcv_A.h/CanTrcv_B.h. – for services and type definitions of the CAN Transceiver Driver
- Dem.h – for services of the Diagnostic Event Manager (DEM)
- Det.h – for services of the Development Error Tracer (DET - optional)
- EcuM.h – for callout declarations of the ECU State Manager
- ComStack_Types.h for COM related type definitions
- 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

CANIF208: The CAN Interface shall include all headers files <Module>_CanIf.h of those upper layer modules, from which declarations of only CAN Interface related API services or type definitions are needed:

- PduR_CanIf.h – for services and callout declarations of the PDU Router

CANIF233: The CAN Interface shall include all header files <Module_Cbk.h>, in which the callback functions called by the CAN Interface at the upper layers are declared:

- CanSM_Cbk.h – for callout declarations of the CAN State Manager
- CanNm_Cbk.h – for callout declarations of the CanNm
- CanTp_Cbk.h – for callout declarations of the CanTp

CANIF280: The CAN Interface shall include all header files <Module_Cfg.h>, which contains the confiuguration data used by the CAN Interface:

- Can_A_Cfg.h/Can_B_Cfg.h – for configuration data of the CAN Driver
- CanTrcv_A_Cfg.h/CanTrcv_B_Cfg.h. – for configuration data of the CAN Transceiver Driver
- 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)

CANIF150: The CAN Interface shall include the file Dem.h. By this way reporting production errors as well as the required Event Id symbols are included. This specification defines the name of the Event Id symbols (see CANIF207), which are provided by XML to the DEM configuration tool. The DEM configuration tool assigns ECU dependent values to the Event Id symbols and publishes the symbols in Dem_IntErrId.h.

CANIF278: The CAN Interface shall include the file MemMap.h in case the mapping of code and data to specific memory sections via memory mapping file is needed for CAN Interface implementation.

6 Requirements traceability

Document: General Requirements on Basic Software Modules [3]

CANIF148:

Requirement	*Satisfied by*
[BSW00344] Reference to link-time configuration	CANIF228
[BSW00404] Reference to post build time configuration	CANIF228
[BSW00405] Reference to multiple configuration sets	CANIF002
[BSW00345] Pre-Build Configuration	Fulfilled by configuration parameter definitions in chapter 10. The configuration parameters are described in a general way.
[BSW159] Tool-based configuration	CANIF104
[BSW167] Static configuration checking	CANIF131
[BSW171] Configurability of optional functionality	Fulfilled by configuration parameter definitions in chapter 10. The configuration parameters are described in a general way.
[BSW170] Data for reconfiguration of SW-components	Not applicable (no interface to AUTOSAR SW Components)
[BSW00380] Separate C-Files for configuration parameters	CANIF151
[BSW00419] Separate C-Files for pre-compile time configuration parameters	CANIF151
[BSW00381] Separate configuration header file for pre-compile time parameters	CANIF122
[BSW00412] Separate H-File for configuration parameters	CANIF122
[BSW00383] List dependencies of configuration files	CANIF141, CANIF066
[BSW00384] List dependencies to other modules	CANIF193, CANIF034
[BSW00387] Specify the configuration class of call-out function	Fulfilled by API definitions in chapter 8.
[BSW00388] Introduce containers	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00389] Containers shall have names	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00390] Parameter content shall be unique within the module	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00391] Parameter shall have unique names	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00392] Parameters shall have a type	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00393] Parameters shall have a range	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00394] Specify the scope of the parameters	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00395] List the required parameters (per parameter)	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00396] Configuration classes	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00397] Pre-compile-time parameters	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00398] Link-time parameters	Fulfilled by configuration parameter definitions in chapter 10.

[BSW00399] Loadable Post-build time parameters	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00400] Selectable Post-build time parameters	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00402] Published information	Fulfilled by sequence diagrams in chapter 9.
[BSW00375] Notification of wake-up reason	CANIF013
[BSW101] Initialization interface	CANIF001
[BSW00416] Sequence of Initialization	Not applicable (no initialization dependencies for this module)
[BSW00406] Check module initialization	Fulfilled by API definitions in chapter 8.
[BSW168] Diagnostic Interface of SW components	Not applicable (this module does not support a special diagnostic interface)
[BSW00407] Function to read out published parameters	CANIF158
[BSW00423] Usage of SW-C template to describe BSW modules with AUTOSAR Interfaces	Not applicable (this module does not provide an AUTOSAR interface)
[BSW00424] BSW main processing function task allocation	Not applicable (requirement on system design, not on a single module)
[BSW00425] Trigger conditions for schedulable objects	Not applicable (requirement on system configuration, not on a single module)
[BSW00426] Exclusive areas in BSW modules	Not applicable (no exclusive areas specified for this module)
[BSW00427] ISR description for BSW modules	Not applicable (this module does not provide any ISRs)
[BSW00428] Execution order dependencies of main processing functions	Fulfilled by description of scheduled functions in chapter 8.5
[BSW00429] Restricted BSW OS functionality access	Not applicable (this module doesn't use any OS objects or services)
[BSW00431] The BSW Scheduler module implements task bodies	Fulfilled by API definitions in chapter 8.
[BSW00432] Modules should have separate main processing functions for read/receive and write/transmit data path	CANIF128
[BSW00433] Calling of main processing functions	Not applicable (requirement on the BSW scheduler module)
[BSW00434] The Schedule Module shall provide an API for exclusive areas	Not applicable (requirement on the BSW scheduler module)
[BSW00336] Shutdown interface	Not applicable (architecture decision)
[BSW00337] Classification of errors	CANIF017
[BSW00338] Detection and Reporting of development errors	CANIF019
[BSW00369] Do not return development error codes via API	CANIF018
[BSW00339] Reporting of production relevant error status	CANIF020
[BSW00417] Reporting of Error Events by Non-Basic Software	Not applicable (this is a basic software module)
[BSW00323] API parameter checking	CANIF022
[BSW004] Version check	CANIF021
[BSW00409] Header files for production code error IDs	CANIF153
[BSW00385] List possible error notifications	CANIF207

[BSW00386] Configuration for detecting an error	CANIF018, CANIF019, CANIF156
[BSW161] Microcontroller abstraction	CANIF035, CANIF036, CANIF143
[BSW162] ECU layout abstraction	CANIF035, CANIF036, CANIF143
[BSW005] No hard coded horizontal interfaces within MCAL	CANIF133
[BSW00415] User dependent include files	CANIF208
[BSW164] Implementation of interrupt service routines	CANIF006, CANIF007
[BSW00325] Runtime of interrupt service routines	CANIF098, CANIF135 The runtime is not totally under control of the CAN Interface, because they are called to the upper layers.
[BSW00326] Transition from ISRs to OS tasks	Not applicable (When a transition from ISR to OS task is done, it will be defined in COM Stack SWS)
[BSW00342] Usage of source code and object code	CANIF228 (post build configuration)
[BSW00343] Specification and configuration of time	Not applicable (no internal scheduling policy)
[BSW160] Human-readable configuration data	Fulfilled by configuration parameter definitions in chapter 10. The configuration parameters are described in a general way.
[BSW007] HIS MISRA C	Not applicable (requirement on implementation, not on specification)
[BSW00300] Module naming convention	Fulfilled by API definitions in chapter 8.
[BSW00413] Accessing instances of BSW modules	Fulfilled by API definitions in chapter 8.
[BSW00347] Naming separation of different instances of BSW drivers	CANIF028
[BSW00305] Self-defined data types naming convention	Fulfilled by type definitions in chapter 8.2.
[BSW00307] Global variables naming convention	Not applicable (requirement on implementation, not on specification)
[BSW00310] API naming convention	Fulfilled by API definitions in chapter 8.
[BSW00373] Main processing function naming convention	CANIF004
[BSW00327] Error values naming convention	CANIF120
[BSW00335] Status values naming convention	CANIF136, CANIF137, CANIF138
[BSW00350] Development error detection keyword	CANIF019
[BSW00408] Configuration parameter naming convention	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00410] Compiler switches shall have defined values	Fulfilled by configuration parameter definitions in chapter 10.
[BSW00411] Get version info keyword	CANIF158
[BSW00346] Basic set of module files	CANIF141
[BSW158] Separation of configuration from implementation	CANIF141
[BSW00314] Separation of interrupt frames and service routines	Not applicable (this module does not provide any ISRs)
[BSW00370] Separation of call-out interface from API	CANIF141
[BSW00435] Module Header File Structure for the Basic Software Scheduler	CANIF241
[BSW00436] Module Header File Structure for the	

Basic Software Memory Mapping	
[BSW00370] Separation of call-out interface from API	CANIF141
[BSW00348] Standard type header	CANIF142
[BSW00353] Platform specific type header	CANIF142 (automatically included with Standard types)
[BSW00361] Compiler specific language extension header	CANIF142 (automatically included with Standard types)
[BSW00301] Limit imported information	CANIF141
[BSW00302] Limit exported information	
[BSW00328] Avoid duplication of code	Not applicable (requirement on implementation, not on specification)
[BSW00312] Shared code shall be reentrant	CANIF065, CANIF077
[BSW006] Platform independency	CANIF143
[BSW00357] Standard API return type	Fulfilled by API definitions in chapter 8.32.
[BSW00377] Module Specific API return type	CANIF136, CANIF137, CANIF138
[BSW00304] AUTOSAR integer data types	Fulfilled by type and API definitions in chapter 8.1 and 8.2
[BSW00355] Do not redefine AUTOSAR integer data types	Fulfilled by type and API definitions in chapter 8.1 and 8.2
[BSW00378] AUTOSAR Boolean type	Not applicable (no Boolean types used)
[BSW00306] Avoid direct use of compiler and platform specific keywords	Not applicable (requirement on implementation, not on specification)
[BSW00308] Definition of global data	Not applicable (requirement on implementation, not on specification)
[BSW00309] Global data with read-only constraint	Not applicable (requirement on implementation, not on specification)
[BSW00371] Do not pass function pointers via API	Fulfilled by API definitions in chapter 8.3
[BSW00358] Return type of init() functions	CANIF001
[BSW00414] Parameter of init function	CANIF001
[BSW00376] Return type and parameters of main processing functions	CANIF004
[BSW00359] Return type of call-out functions	Fulfilled by call-out APIs in chapter 8.4.
[BSW00360] Parameters of call-out functions	Fulfilled by call-out APIs in chapter 8.4.
[BSW00329] Avoidance of generic interfaces	No generic interface used The content of functions might be configuration dependent. The scope of function is always defined
[BSW00330] Usage of macros instead of functions	Not applicable (requirement on implementation, not on specification)
[BSW00331] Separation of error and status values	CANIF120 CANIF136, CANIF137, CANIF138
[BSW009] Module User Documentation	Fulfilled by the complete documentation.
[BSW00401] Documentation of multiple instances of configuration parameters	Fulfilled by configuration parameter definitions in chapter 10.
[BSW172] Compatibility and documentation of scheduling strategy	Not applicable (no internal scheduling policy)
BSW010] Memory resource documentation	Not applicable (requirement on implementation, not on specification)
[BSW00333] Documentation of call-out function context	Fulfilled by call-out APIs in chapter 8.4.

[BSW00374] Module vendor identification	CANIF016
[BSW00379] Module identification	CANIF016
[BSW003] Version identification	CANIF016
[BSW00318] Format of module version	CANIF016
[BSW00321] Enumeration of module version numbers	CANIF016
[BSW00341] Microcontroller compatibility documentation	Not applicable (no microcontroller dependent module)
[BSW00334] Provision of XML file	Not applicable (requirement on implementation, not on specification)

Document: Requirements on CAN [4]

Requirement	***Satisfied by***
[BSW01033] Basic Software General Requirements	CANIF148
[BSW01125] Data throughput read direction	CANIF112
[BSW01126] Data throughput write direction	CANIF161
[BSW01139] CAN Controller specific Initialization	CANIF002
[BSW01129] Receive Data Interface for CAN Interface and CAN Driver Module	CANIF196
[BSW01121] Interfaces of the CAN Interface module	CANIF034, CANIF266
[BSW01014] Network configuration abstraction	See all APIs in chapter 8.3 Function definitions
[BSW01001] HW independence	CANIF023
[BSW01015] Network Database Information Import	CANIF104
[BSW01016] Interface to CAN Driver configuration	CANIF066
[BSW01017] Polling/interrupt mode	CANIF029
[BSW01018] Software Filter	CANIF030
[BSW01019] DLC Check configuration	CANIF031
[BSW01020] Tx Buffer configuration	CANIF071
[BSW01021] CAN Interface Module Power-On Initialization	CANIF001
[BSW01022] Dynamic selection of static configuration sets	CANIF092
[BSW01023] Power-On Initialization Sequence	CANIF032
[BSW01002] Rx PDU dispatching	CANIF024
[BSW01003] Reception indication dispatcher	CANIF012
[BSW01114] Data Consistency of transmit L-PDUs	CANIF033
[BSW01004] Software Filtering for L-PDU reception	CANIF025
[BSW01005] DLC check for L-PDU reception	CANIF026
[BSW01006] Rx L-PDU enable/disable	CANIF096
[BSW01007] Tx L-PDU dispatching	CANIF028
[BSW01008] Transmission request service	CANIF005
[BSW01009] Transmission confirmation service	CANIF007
[BSW01011] Tx buffering	CANIF068
[BSW01013] Tx L-PDU enable/disable service	CANIF096
[BSW01027] CAN controller Mode Select service	CANIF003
[BSW01028] CAN controller State Service	CANIF093
[BSW01032] Wake-up Notification	CANIF013
[BSW01061] Dynamic Tx Handles	CANIF185
[BSW01024] DLC Error Notification	Skipped to due bug #14340

[BSW01029] Bus-off notification	CANIF014
[BSW01130] Read Status Interface of CAN Interface	CANIF200
[BSW01131] Mixed mode of notification and polling mechanism	CANIF197, CANIF203
[BSW01136] Notification of first received CAN message	CANIF182
[BSW01129] Receive Data Interface for CAN Interface	CANIF194
[BSW01140] Support of Standard and Extended Identifiers	CANIF281
[BSW01141] Support of both Standard and Extended Identifiers on one network (optional feature)	CANIF243, CANIF261

7 Functional specification

7.1 General functionality

CANIF041: The services of the CAN Interface can be divided into the following main groups:

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

CANIF042: Possible applications:

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

2. Polling mode
   The CAN Driver is triggered by the BSW Scheduler 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 CAN Interface is notified by the CAN Driver about events (reception, transmission, BusOff), that occurred in one of the CAN controllers, equally to the interrupt driven operation. The CAN Driver is responsible for the update of the corresponding information which belongs to the occurred event in the CAN controller, for example reception of an L-PDU.

3. Mixed mode: interrupt and polling driven CAN Driver
   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 an unique interface, which is valid for all three types of operation modes. Summarized the CAN Interface works in the same why, 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 CAN Interface.

7.2 Hardware object handles

CANIF023: Hardware object handles (HOH) for transmission (HTH) as well as for reception (HRH) represent an abstract reference to a CAN RAM structure that contains CAN related parameters such as CAN ID, DLC and data. Based on this CAN hardware buffer abstraction each hardware object is referenced in the CAN Interface independent of the CAN hardware buffer layout. The HOH is used as a parameter in the calls of the CAN Driver interface services and is provided by the CAN Driver's configuration and used by the CAN Driver as identifier for communication buffers of the CAN mailbox.
The CAN Interface acts only as user of the Hardware object handle but does not interpret it on the basis of hardware specific information. The CAN Interface therefore remains independent of hardware.

CANIF043: Each CAN controller can provide a pool of hardware objects in the CAN mailbox. These can be logically linked to form one entire pool of hardware objects (multiplexed hardware objects).

CANIF044: Two types of Hardware object handles are used in the CAN Interface to enable access to the CAN Driver: Hardware Receive Handle (HRH) and Hardware Transmit Handle (HTH).

CANIF291: The HRH represents a logical reception unit. This unit consists of just one hardware object used for the reception. HRH, which is dependent on the settings of the acceptance masks, is marked as a BasicCAN or a FullCAN receive unit which determines the usage of software filtering. Each CanRxPduId is assigned to one single HRH. Thus if multiple HRHs are used, each HRH belongs to a single or fixed group of CanRxPduId.

CANIF292: The HTH represents a hardware objects configured for transmission purposes and for the corresponding CAN controller. Each CanTxPduId is assigned to one single HTH. Thus if multiple HTHs are used, each HTH belongs to a single or fixed group of CanTxPduIds.

CANIF115: All HRH and HTH handles of one CAN Driver has an own numbering area. The dedicated HRH and HTHs handles are derived from the configuration set of the CAN Driver(s). The definition of HTH/HRH inside the numbering area and hardware objects is up to the CAN Driver. It has to be ensured by configuration, that no overlapping of several numbering areas of multiple CAN Drivers is allowed.

CANIF123: The HRH can be configured to receive
- one single CAN ID (FullCAN)
- a group of single CAN IDs (BasicCAN)
- a range/area of CAN IDs (BasicCAN) or
- all CAN IDs.

7.3 Static CAN L-PDU handles

CANIF045: The CAN Interface offers general access to the CAN L-PDU related data for upper layers. This access is achieved by the L-PDU handle. The L-PDU handle refers to data structures, which consists of attributes describing the L-PDU. There are two kinds of attributes: CAN PCI and CAN Interface specific attributes.

CAN Interface specific attributes	*CAN Protocol Control Information (PCI)*
Method of SW filtering	CAN Identifier (ID)
Direction of L-PDU (Tx, Rx)	Data Length Code (DLC)
Physical channel (refer to [7.20.1 PDU channel groups])	Reference to the data (SDU)
HTH/HRH of the CAN controller	
Target ID for the corresponding upper layer	
Type of receive L-PDU (FullCAN, BasicCAN)	

Table 1 Attributes used in CAN Interface

For the optimization of further processing a part of this information can be represented by the L-PDU handle itself.

CANIF046: Each L-PDU and thus each L-PDU handle is dedicated to one CAN controller only. This relation is used in order to ensure the correct dispatch at transmission and reception. In this manner the CAN Interface is able to reconstruct the CAN contoller from the L-PDU handle.

CANIF047: The CAN Interface supports activation and deactivation of all L-PDUs belonging to one CAN network for transmission as well as for reception (CANIF027). For L-PDU mode control refer to [7.20 PDU channel mode control].

CANIF048: Each L-PDU handle is associated with an upper layer in order to ensure the correct dispatch service during reception, transmission confirmation and data access.
Each upper layer can use the L-PDU handles to serve different CAN controllers simultaneously.

CANIF236: According to the PDU architecture defined for the entire AUTOSAR communication stack (see [2] Layered Software Architecture), the usage of L-PDUs is split in two different ways:

a) For transmission request and transmission/reception polling API the upper layer uses the CAN L-PDU Id defined by the CAN Interface as parameter.
b) For all call-out APIs, which are invoked by the CAN interface at the upper layers, the CAN Interface passes the target PDU-Id defined by each upper layer as parameter.

In principle: the caller must use the defined target PDU Id of the callee.

CANIF239: If power on initialization is not performed, no L-PDUs are transmitted and DET is informed. Thus no uninitialized data can be transmitted on the network.

7.4 Dynamic CAN transmit L-PDU handles

CANIF185: Dynamic transmit L-PDUs make possible to reconfigure during runtime the CAN identifier to be used for the corresponding L-PDU handle.

CANIF186: The maximum number of dynamic transmit L-PDU handles shall be set pre-compile time by the configuration parameter `CANIF_NUMBER_OF_DYNAMIC_CANTXPDUIDS`. This parameter can be updated during post-build time.

CANIF187: The confirmation notification belongs to the L-PDU handle, thus it can not be changed. The data length code (DLC) and the pointer to the data buffer is determined by the upper layer during `CanIf_Transmit()`.

CANIF188: The CAN identifier shall be reconfigured by `CanIf_SetDynamicTxId()`. Most significant bit of CAN-ID must be set to one while passing extended CAN-ID to support mixed mode of operation.

Hint:This function may not be interrupted by `CanIf_Transmit()` in case of the same L-PDU handle is affected. This way ensures data integrity of the CAN identifier. This has to be ensured by the upper layer of CanIf.

CANIF238: The CAN identifiers of the dynamic transmit CAN L-PDUs shall only be initialized by `CanIf_Init()`. `CanIf_InitController()` has no effect on dynamic transmit L-PDUs to avoid re-initialization of dynamic transmit L-PDUs' CAN identifiers by upper layer users.

7.5 Physical channel view

CANIF049: The CAN Interface API represents a view on all managed physical CAN channels. Those are used by the CAN State Manager to provide a network view to the COM Manager used to perform wakeup and sleep request for all physical channels connected to a single network. 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.

CANIF170: The CAN Interface passes status information provided by the CAN Driver and CAN Transceiver Driver separately for each physical channel as status information for the CAN State Manager (refer to 8.5.3 Configurable interfaces), which has to manage the network specific operation mode.

CANIF272: During this notification process the CAN Interface passes the original CAN controller or CAN Transceiver parameter to the CAN State Manager.

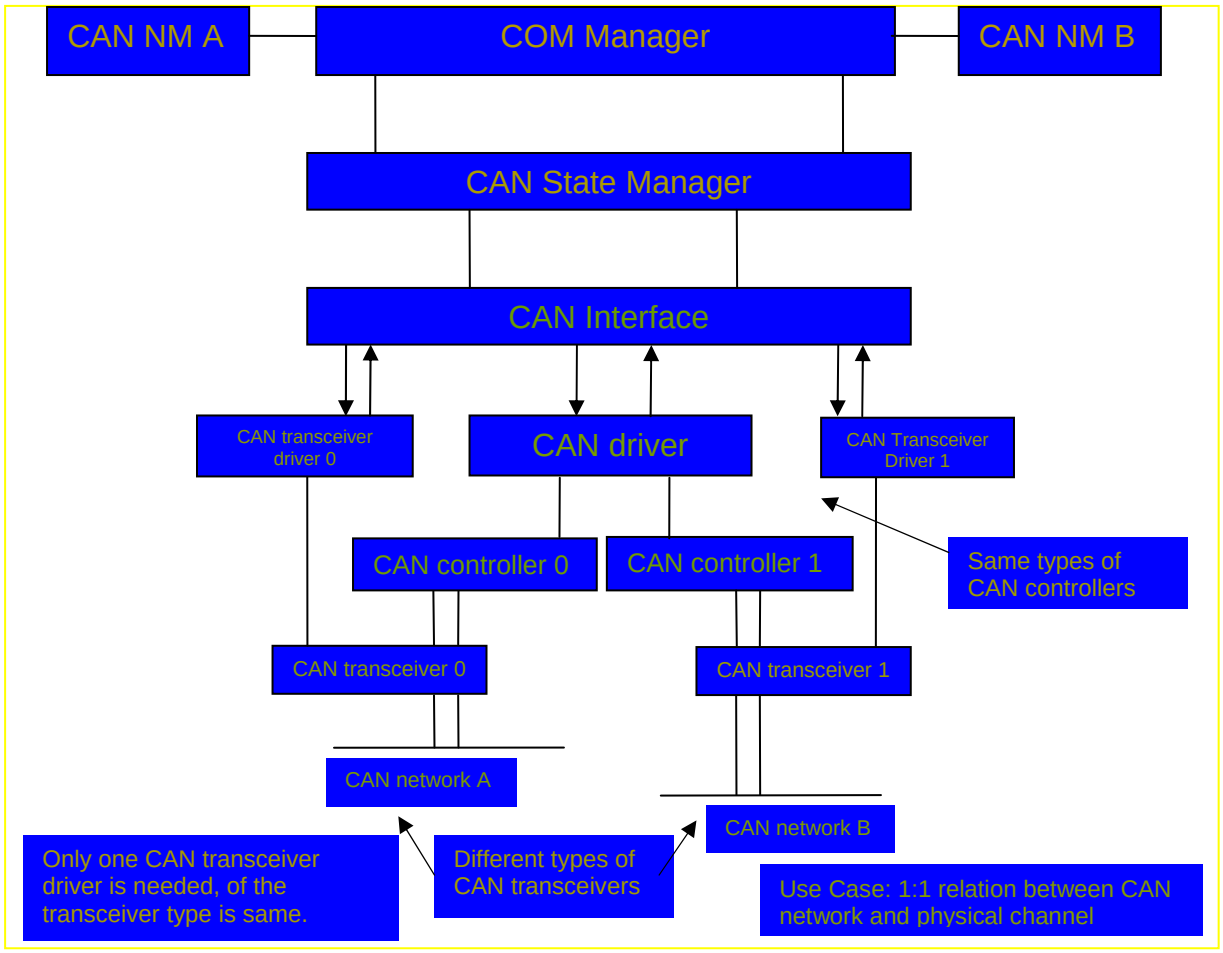

Figure 4: Physical channel view definition example A

CANIF105: The CAN Interface supports multiple physical CAN channels. These have to be distinguished by the CAN State Manager for network control. The CAN Interface API provides request and read control for multiple underlying physical CAN channels.

CANIF169: Moreover the CAN Interface does not consider dedicated types of CAN physical layers (i.e. low-speed CAN or high-speed CAN), to which one or multiple CAN controllers are connected.

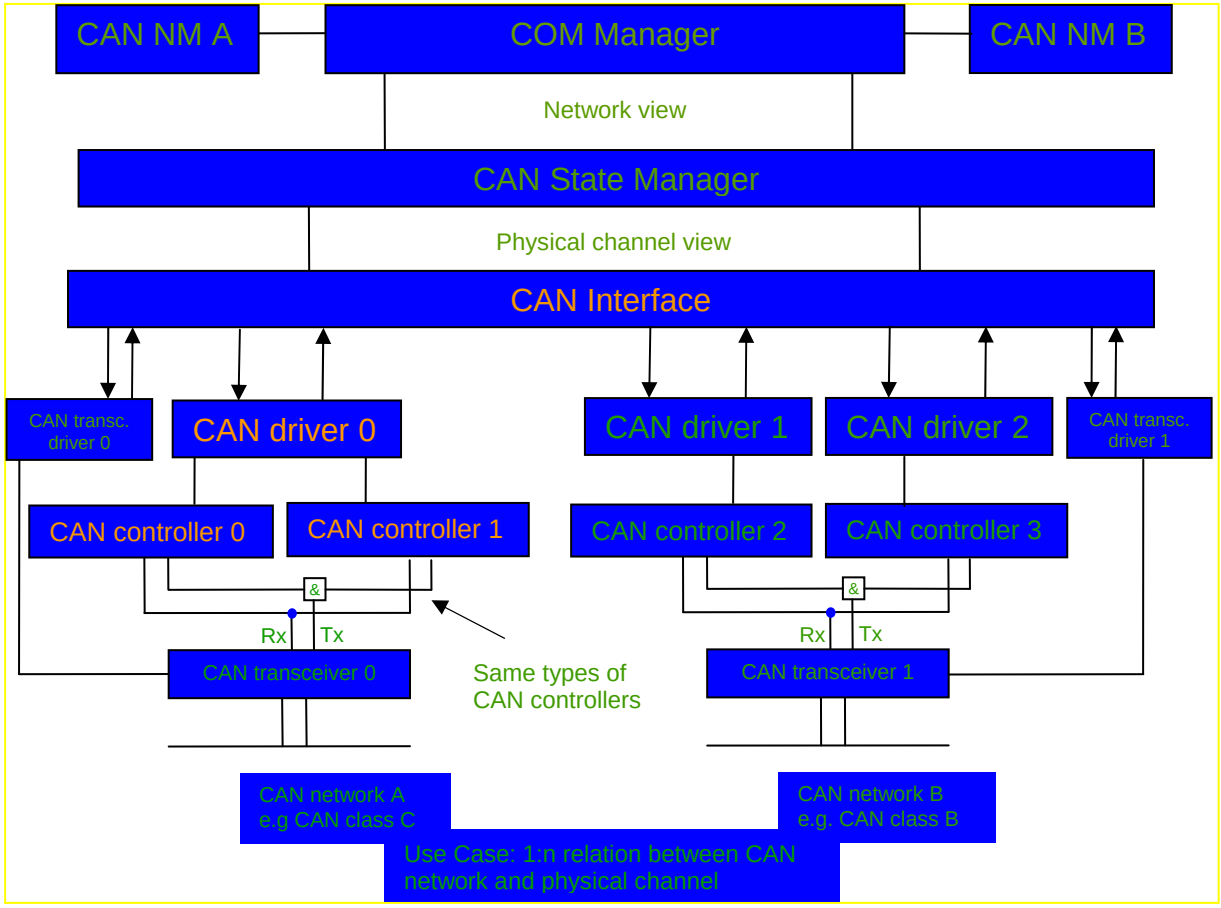

Figure 5: Physical channel view definition example B

7.6 CAN hardware unit

CANIF209: The CAN hardware unit combines one or multiple CAN controllers 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.
If different types of CAN controllers are used, also different types of CAN Drivers have to be applied with a unified API to the CAN Interface. The CAN Interface collects information about the types and number of CAN controllers and their hardware objects in its mapping tables at configuration time. This allows transparent and hardware independent access to the CAN controller from upper layers (CANIF023) (refer to [7.25 Multiple CAN Driver support]).

CANIF210: The following figure shows a CAN hardware unit consisting of two CAN controllers of the same type connected to two physical channels:

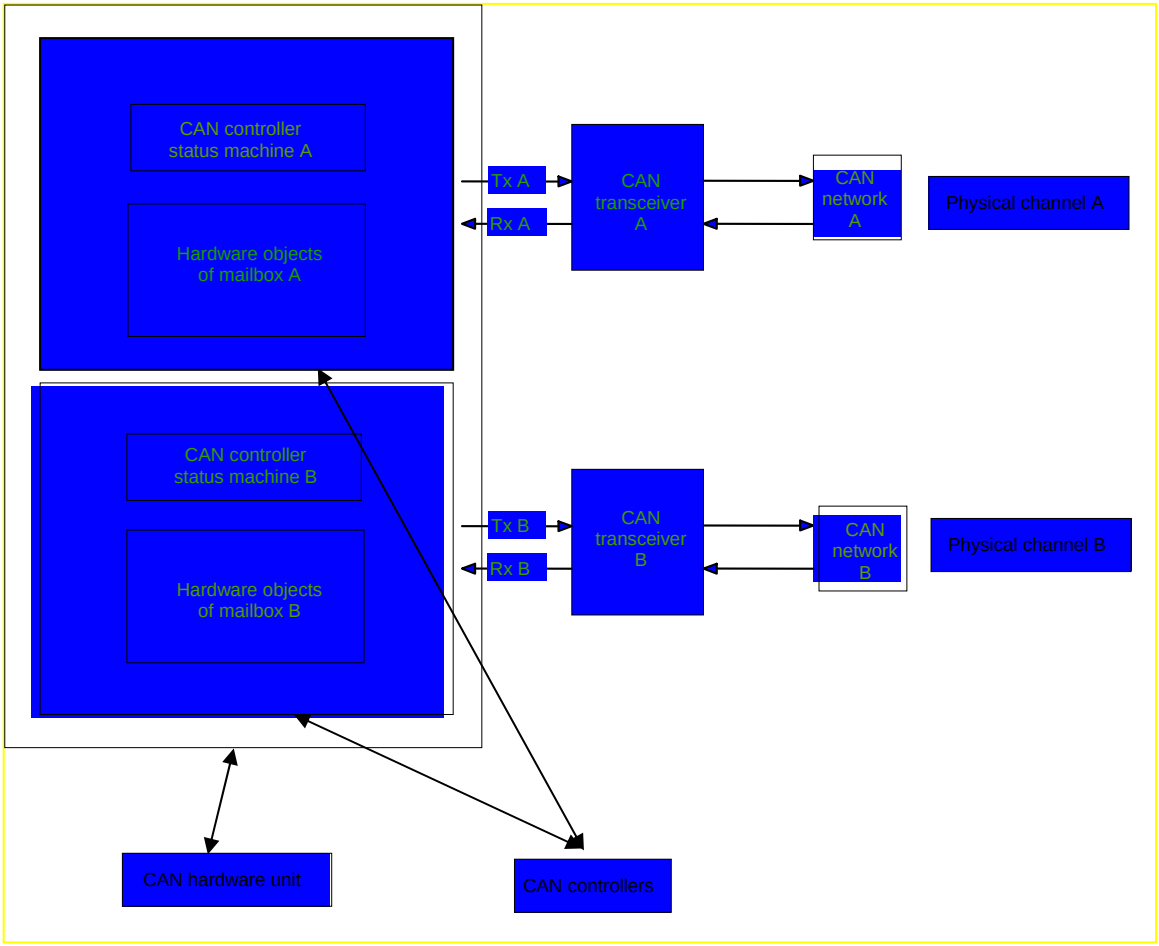

Figure 6 Typical CAN hardware unit

7.7 BasicCAN and FullCAN reception

CANIF050: An appropriately configured hardware object for FullCAN operation only enables transmission or reception of a single CAN ID. Accordingly BasicCAN operation of one hardware object enables to send or receive a range of CAN IDs.

CANIF164: The hardware acceptance filter is a significant attribute to be configured of each hardware object. It is used in combination with subsequent software filtering at BasicCAN reception to filter out receive L-PDUs, which are not part in the list of predefined receive L-PDUs of the local ECU. For FullCAN reception the hardware acceptance filter must be configured for full match to the received CAN Identifier.

CANIF051: The CAN Interface distinguishes between BasicCAN and FullCAN handling for activation of software filtering. Thus it derives the corresponding configuration of the CAN hardware objects by CAN Driver's configuration setup. It defines the number and order of transmit and receive hardware objects, configures the hardware objects for optimal BasicCAN/FullCAN reception and transmission and allows free setting of acceptance filters for each BasicCAN objects.

CANIF211: The main difference between BasicCAN and FullCAN operation is in the need of a Software Filtering mechanism (CANIF025) at reception of incoming PDUs over BasicCAN hardware objects. At this time the appropriate software filtering algorithm is executed in dependence on whether a PDU reception took place in a receive BasicCAN hardware object. At configuration time the relation between FullCAN/BasicCAN reception and HRH is stored in the CAN Driver's public configuration setup. The CAN Interface thus derives BasicCAN or FullCAN reception strategy based on the HRH passed by the API call-out `CanIf_RxIndication()`.

CANIF108: Using an appropriate hardware acceptance filter configuration allows using BasicCAN receive objects for special upper layer use cases, i.e. diagnostic L-PDU reception. In this case the CAN Interface detects a reception event in a BasicCAN object configured for a special L-PDU or a range of special L-PDUs by the HRH of the `CanIf_RxIndication()` call-out service. This functionality can only be established, if the CAN controller mailbox is able to be configured appropriately. Therefore this functionality depends on the CAN controller type and its configuration.

CANIF107: BasicCAN operation is optional due to it is not necessary, if there are less receive L-PDUs than available receive FullCAN Objects.

CANIF165: Usage of multiple BasicCAN and FullCAN receive objects are supported, if provided by the underlying CAN controllers. BasicCAN and FullCAN objects can be used in parallel..

CANIF243: Basically the CAN Interface can be configured to support reception either of 11 bit StandardCAN or 29 bit ExtendedCAN CAN identifiers on one network. While passing CAN-ID to CAN Driver during CanIf_Transmit, most significant bit of the extended CAN-ID shall be set to one to distinguish between standard and extended.

CANIF281: If needed, StandardCAN and ExtendedCAN shall be also supported as mixed mode operation, whereas both identifier types can be used mixed at the same time on the network. By that way, the BasicCAN/FullCAN hardware objects have to be separately configured for either StandardCAN or ExtendedCAN operation. This is an optional feature. This feature can be realized by different variants of implementations, no configuration option is available.

CANIF261: To support the usage of cheap BasicCAN CAN controllers mixed mode operation is supported also for a single receive BasicCAN hardware object. In that case the software search algorithm (see 7.21 Software receive filter) must be able to deal with both type of CAN identifiers.

7.8 Initialization

CANIF032: The CAN Interface provides different API services for both global and controller specific CAN controller initialization. CAN controller specific initialization way is necessary in order to ensure different startup behaviors of CAN controllers that are connected to different CAN networks.

CANIF085: The initialization process is executed by the call of `CanIf_Init()` for global initialization. During the initialization process the global variables and data structures are initialized including flags and buffers only. The CAN Drivers and the CAN Transceiver Drivers are initialized separately.

CANIF293: All CAN controllers and all the configured buffers in the CAN Interface of all Tx/Rx L-PDUs can be (re-)initialized by the call of `CanIf_InitController()`. That means, that this call itself initiates also the controller specific initialization of underlying CAN Drivers. Subsequently the CAN Interface calls the corresponding CAN Driver initialization services.
The API service `CanIf_InitController()` makes it possible to change the setup of all CAN controller of one CAN network after initialization. For those use cases multiple CAN controller configurations have to be set up during configuration time for each CAN controller.

CANIF086: The CAN Interface expects, that after Initialization of the CAN Driver the CAN controller shall remain in STOPPED Mode. In this mode it is neither able to transmit nor receive CAN L-PDUs.

CANIF092: Initialization processes shall only take place in STOPPED and UNINT mode. UNINIT mode is left only if global initialization once after reset is requested, whereas in STOPPED mode both initialization APIs for global initialization can be used (refer to [7.19.2.1CANIF_CS_UNINIT]). If initialization is performed in STARTED mode, the CAN Interface will perform the transition to STOPPED mode.

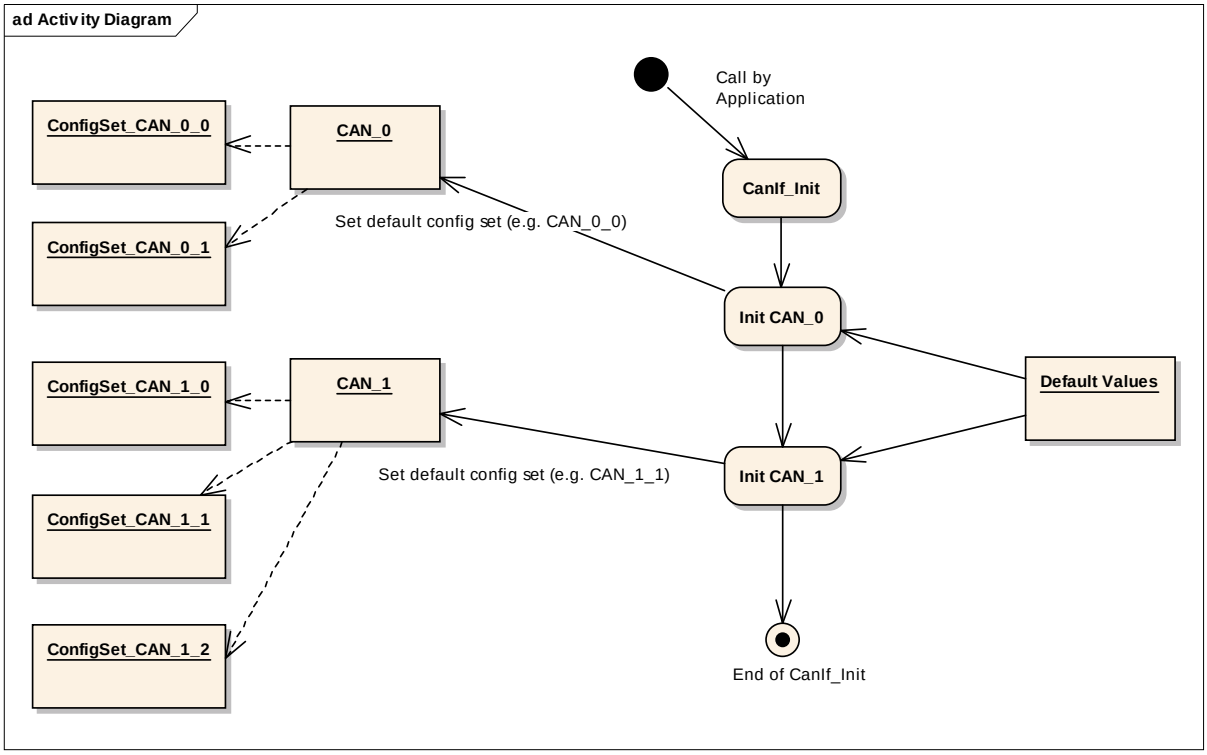

Figure 7 Controller specific initialization of CAN Interface and CAN Driver

The figure above shows the relationship between configuration sets CAN_0/CAN_1 referring to different CAN Drivers and configuration sets CAN_0_0, CAN_0_1 and CAN_1_0 CAN_1_1 referring to single CAN controllers.

7.9 Transmit data flow

CANIF160: The transmission API of the CAN Interface is based on L-PDU handles. Each transmit request is initiated by the calling of the CAN Interface service `CanIf_Transmit()`. The access to the L-PDU specific data is organized by

- transmit L-PDU Handle and
- a reference to a structure with PDU related data: SDU length and pointer to the L-SDU.

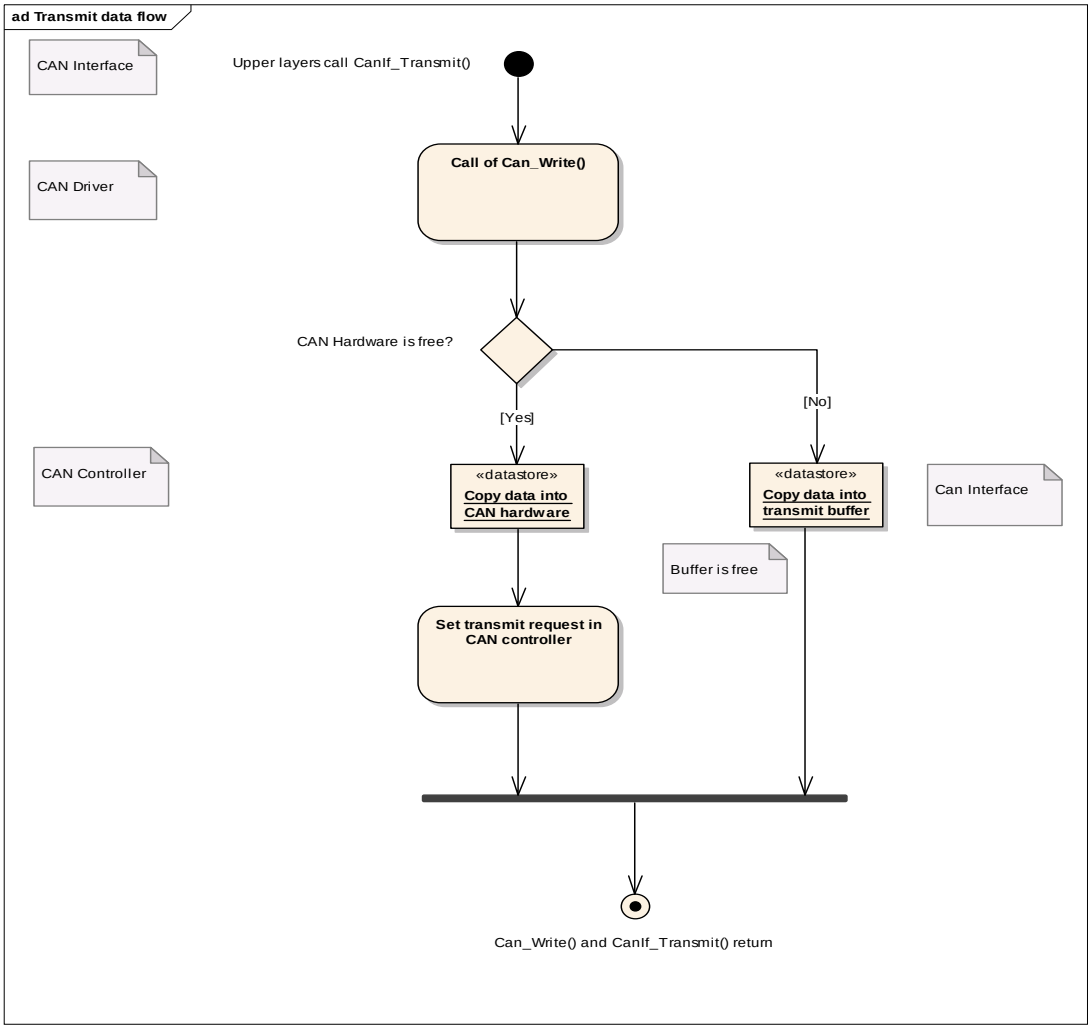

Figure 8 Transmit data flow

CANIF161: The CAN Interface holds all information about hardware objects configured for transmission purposes. It calls an interface service `Can_Write()`, provided by the CAN Driver, with the transmit Hardware object handle as a

parameter. The requested CAN controller is identified by this handle. The `Can_Write()` service carries out the hardware dependent operations and sets up the transmit request in the CAN controller. If no free hardware objects are available at time of the transmit request, the CAN Driver `Can_Write()` service returns `CAN_BUSY` and the transmit request is inserted in the transmit buffer. If no resources are available in the transmit buffer (refer to [7.12 Transmit buffering]) or the CAN controller is in STOP mode, the transmit request `CanIf_Transmit()` returns `E_NOT_OK` and the production error `CANIF_E_FULL_TX_BUFFER` respectively `CANIF_E_STOPPED` is raised. In this case the upper layer is responsible for the repeating of the transmit request.

CANIF162: A successful transmission will be indicated to the upper layer by call of the appropriate upper layer confirmation call-out service `<User_TxConfirmation>()`.
The reference to the L-PDU specific data is organized via pointer on an L-PDU structure used as a parameter. This structure contains L-PDU specific data like L-SDU length and pointer on L-SDU buffer. This interface enables design with central placed L-SDU buffers in the CAN Interface as well as with distributed placed L-PDU buffers in the upper layers.

CANIF163: The CAN Interface temporarily stores L-PDUs to be transmitted only in case of locked CAN controller hardware.

7.10 Transmit request

The transmit request API (CANIF005) is a common interface for upper layers to send PDUs on the CAN network. The upper communication layers initiate the transmission only via the CAN Interface services without direct access to the CAN Driver. The initiated transmit request is successfully completed, if the CAN Driver could write the L-PDU data into the CAN hardware.

CANIF082: If no hardware resources were available at the time of initiation, the state of the transmit request obtains the state "pending" and the complete L-PDU including the L-SDU is temporarily stored in the CAN Interface. When the previous transmission is completed and hardware resources are released the subsequent transmit requests are carried out. If no hardware and also no software buffers are available the transmit request is rejected immediately (refer to [7.12 Transmit]).

The upper layer uses the CAN Interface service `CanIf_Transmit()` to initiate a transmit request.

The CAN Interface offers two parameters inside the `CanIf_Transmit()` service only to localize the L-SDU buffers in the upper layer and make them not global. These are L-PDU Handle and pointer on the L-PDU Structure which contains all related parameters like SDU length and pointer to the L-SDU buffer.

The CAN Interface performs following actions for L-PDU transmission:

- Request based transmit handling,

- CAN Driver and hardware object routing

The `CanIf_Transmit()` service returns `E_NOT_OK`, if previously the CAN Interface was not initialized or the all transmit hardware objects contains pending transmit requests and no transmit buffering was configured [7.12 Transmit buffering].

7.11 Transmit confirmation

7.11.1 Confirmation after transmission

The upper communication layer may be notified about the performed transmission via the CAN Interface confirmation services after the successful completion of the transmission (CANIF007). A previous transmit request is processed successfully, if at least one remote ECU in the network acknowledges the transmitted CAN Frame in the CAN acknowledge slot. The CAN Interface is notified by the CAN Driver by call of `CanIf_TxConfirmation()`. The call-out service `<User_TxConfirmation>()` implemented by the notified upper layer will subsequently be called by the CAN Interface, if this service is enabled at configuration time for transmission confirmation.

CANIF053: An upper communication layer can be configured to process the confirmations with a single or multiple call-out services for different L-PDUs or groups of L-PDUs. All that services are called by the CAN Interface at confirmation of the corresponding L-PDU transmission. The transmit L-PDU Handle enables the dispatch between different confirmation services associated to the upper layer target. This assignment is made statically during configuration.

CANIF109: A single transmit L-PDU can only be assigned to one single confirmation call-out service Please refer to [10.2.8 CanInterfaceDispatcherConfiguration].

For L-PDU confirmation the CAN Interface Layer performs protocol dispatching for the upper layers (CANIF028).

7.11.2 Confirmation of transmit cancellation

CANIF054: Some CAN controllers provide cancellation of the pending transmit request inside its transmit object of the CAN controller. This is used to prevent inner priority inversion. A pending transmit request within a hardware object is canceled and exchanged by an L-PDU with higher priority. The CAN Driver detects a successful transmit cancellation by a corresponding confirmation interrupt or polling dependent on the used CAN controller. The CAN Interface is informed by the callback notification service `CanIf_CancelTxConfirmation()` and stores the aborted old L-PDU inside its transmit buffer as long it is free .

For more details about transmission cancellation please refer to chapter [7.13 Transmit cancellation].

7.12 Transmit buffering

7.12.1 General behavior

CANIF103: The CAN Interface provides transmit buffers (located in global RAM) to store whole L-PDUs, which are rejected by the CAN Driver at transmission request. The CAN Driver can reject transmission requests, when no transmit hardware resources (CAN RAM) are available within the corresponding CAN controller. No hardware resources are available, if all hardware transmit objects are is use and the priority of the new requested L-PDU is lower than all pending ones in the hardware objects.

CANIF091: During the transmit confirmation the transmit buffer is analyzed handling whether a pending transmit L-PDU is stored. The transmit confirmation handling is executed either inside the confirmation interrupt service routine or at task level during polling by `Can_MainFunction_Transmit()`. If pending transmit L-PDUs are available in the transmit buffer, the CAN Interface initiates a new transmit request. The CAN Driver writes it in the free hardware transmit object.

CANIF068: Generally the transmit buffer consists of single element L-PDU buffers. Each L-PDU buffer (global RAM) is statically assigned to one transmit L-PDU handle at configuration time. In this meaning the CAN Interface stores at maximum only just one element per L-PDU in the transmit buffer. Consequently the recent transmit request of a dedicated L-PDU overwrites old ones. This handling prevents, that newer data stored in the transmit buffer may be overwritten from old ones.
f the order of various transmit requests of different L-PDUs shall be kept, transmit requests of upper layers must be connected to previous transmit confirmation notifications. That means, that a subsequent L-PDU is requested for transmission by the upper layers only, if the transmit confirmation of the previous one was notified by the CAN Interface.

Note: Additionally the order of transmit requests can vary 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 (refer to [8]).

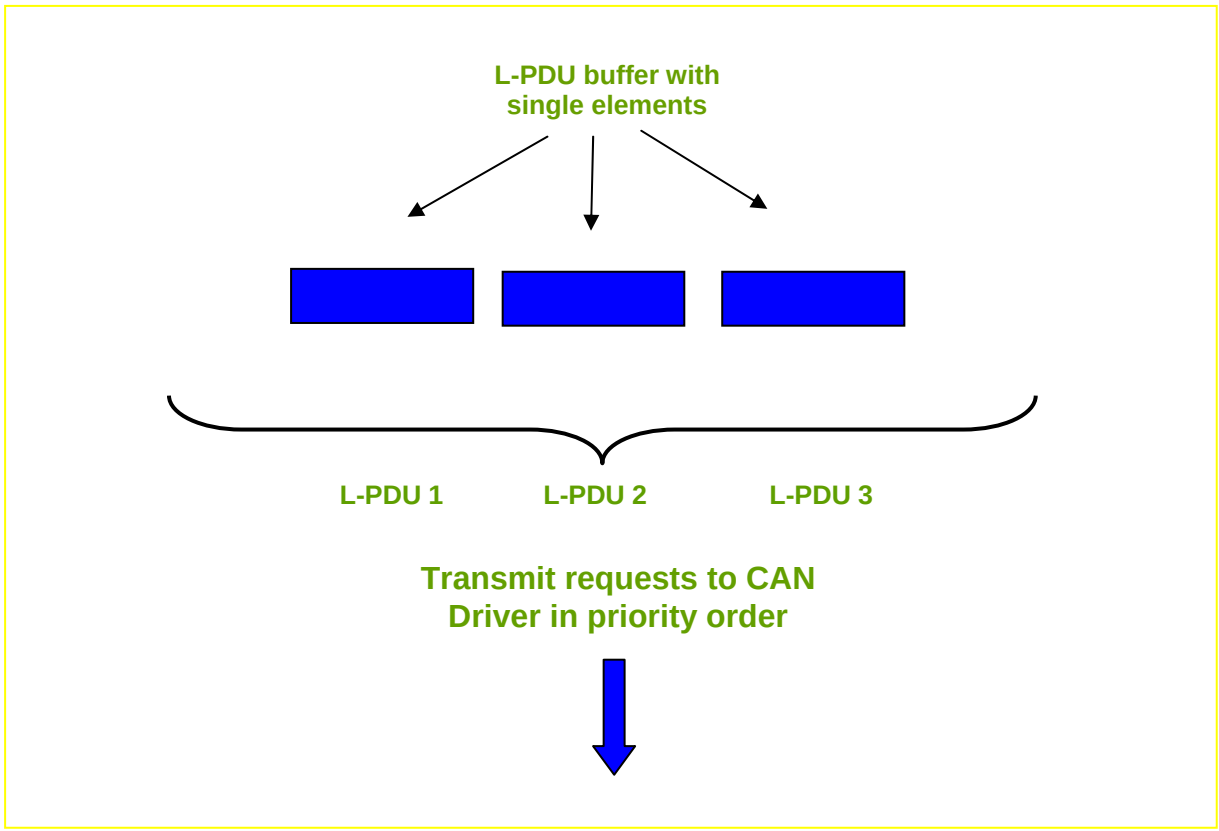

Figure 9 Overview about transmit buffer

CANIF071: The behavior of the transmit buffer differs according to the configuration setup. The following configuration options are provided by the CAN Interface to be able to fulfill all requirements mentioned above.

- Option A: no buffer. Transmit L-PDUs from failed transmit requests are lost. The API `CanIf_Transmit()` returns the value `E_NOT_OK`.

- Option B: L-PDU buffering. One L-PDU per element is stored.

CANIF063: The number of all transmit buffers depends on the number of used transmit L-PDUs defined in the CAN network description file for this ECU. RAM optimization: in case of using several transmit L-PDUs it is recommended not to reserve statically one buffer per L-PDU to save RAM. Thus the total amount of elements is configurable. Are less buffer elements configured than available transmit L-PDUs, the L-PDU oriented buffer elements are dynamically assigned during runtime to the transmit L-PDU pending for transmission. Nevertheless only one element per L-PDU can be stored in the buffers.

CANIF282: Dynamic transmit L-PDUs have to be buffered based on the currently used CAN ID. No overwrite of pending dynamic transmit L-PDUs with the same L-PDU IDs and different CAN IDs may occur.

7.12.2 Buffer characteristics

7.12.2.1 Storage of L-PDUs in the transmit buffer

CANIF113: The CAN Interface tries to store a new transmit L-PDU in the transmit buffer only, if
- the CAN Driver return CAN_BUSY during a call of Can_Write() or
- a pending transmit request was successfully aborted

7.12.2.2 Storage of L-PDUs is prohibited

CANIF069: Whenever
- transmit cancellation is enabled (see chapter [7.13 Transmit cancellation]),
- the CAN Driver notifies the CAN Interface about an aborted Tx L-PDU and
- the same transmit L-PDU from another upper layer's transmit request is already stored in the transmit buffer, the 'old' aborted transmit L-PDU is not stored in the transmit buffer and therefore is lost. Aborted transmit L-PDUs are only stored in the transmit buffer, if the corresponding L-PDU buffer is free. This behavior ensures, that always the most recent data is stored in the transmit buffer.

7.12.2.3 Get L-PDU with the highest priority

CANIF070: The CAN Interface transmits L-PDUs stored in the transmit buffer in priority order per each HTH.

7.12.2.4 Remove transmitted L-PDU

CANIF183: When the highest prior L-PDU stored in Tx buffer has to be transmitted during execution of transmit confirmation and `Can_Write()` returns with success, this L-PDU is removed immediately from the transmit buffer, before the transmit confirmation returns. This behavior simplifies the choice of the new transmit L-PDU stored in the transmit buffer.

7.12.2.5 Initialization of transmit buffers

CANIF184: At `CanIf_Init()`as well as in case of needed controller specific re-initialization `CanIf_InitController()`during the BusOff recovery the initialization of the transmit buffers is processed. This is necessary to prevent transmission of old data after restart of the CAN controller.

7.12.3 Data integrity of transmit buffers

CANIF033: Access events to the transmit buffer like storing a new L-PDU or removing transmitted L-PDU can occur preemptively. Therefore the access to buffers for all transmit L-PDUs takes place in critical sections (refer to CANIF065).

CANIF076: If multiple CAN controllers are used, only the single buffer of the transmit L-PDU to be stored must be locked, because all L-PDU buffers are organized separately. This is valid for use cases with a single or multiple HTHs.

7.13 Transmit cancellation

CANIF083: All pending transmit requests are transmitted in priority order. The PDU priority is implicitly defined by the CAN ID. Other priority definitions are prohibited to avoid priority inversion at transmit request order. The abort of pending transmit L-PDUs within the transmit hardware objects is necessary to avoid inner priority inversion. The mechanism of the transmit processing differs, whether hardware cancellation is supported or not.

7.13.1 Hardware transmit cancellation not supported or not used

CANIF175: The L-PDU request is stored in the temporary single element transmit buffer, if the corresponding CAN hardware transmit buffers are busy. If a free Hardware object is available, the L-PDU is directly transmitted using the `Can_Write()` service of the CAN Driver.
Constraint: If all available hardware objects are busy and a L-PDU is pending for transmission with a higher priority, this L-PDU is delayed until a hardware object is released.

7.13.2 Hardware transmit cancellation supported and used

CANIF176: At time of a new transmit request the CAN Driver checks, whether a free hardware object is available. If all hardware objects are in use, the CAN ID of the requested L-PDU is compared with the CAN ID of all pending L-PDUs in the hardware transmit objects. If the requested L-PDU has a higher priority than any pending one, the transmission of lowest prioritized pending transmit L-PDU is aborted and the new L-PDU is put in the hardware transmit object. The L-PDU to be transmitted is stored in the transmit buffers. The CAN Driver confirms the transmit cancellation by the callback service `CanIf_CancelTxConfirmation()` and passes the old L-PDU back to the CAN Interface's transmit buffer. See UML diagram in chapter [9.6].

CANIF114: In dependence of the used CAN controller and the traffic on the network the cancellation of a pending transmit L-PDU inside a hardware object can occur asynchronously. The transmit buffers are able to distinguish between aborted transmit L-PDUs and new pending transmit L-PDUs. This is necessary to ensure to keep the latest data of several pending transmit L-PDUs with the same L-PDU handle (refer to CANIF113). In that way a successful cancelled L-PDU is lost, when the L-PDU transmit buffer was already occupied by a new L-PDU of the same L-PDU handle.

CANIF084: In case of a transmit confirmation or a transmit cancellation confirmation from the CAN Driver the next stored transmit L-PDU with the highest priority is sent out (see chapter [7.12 Transmit buffering]). Because of pre-emptive events of storing and processing of pending transmit requests in the transmit confirmation context the transmit request service is called re-entrant. Therefore the CAN Interface locks all critical accesses by an internal semaphore mechanism or interrupt locks.

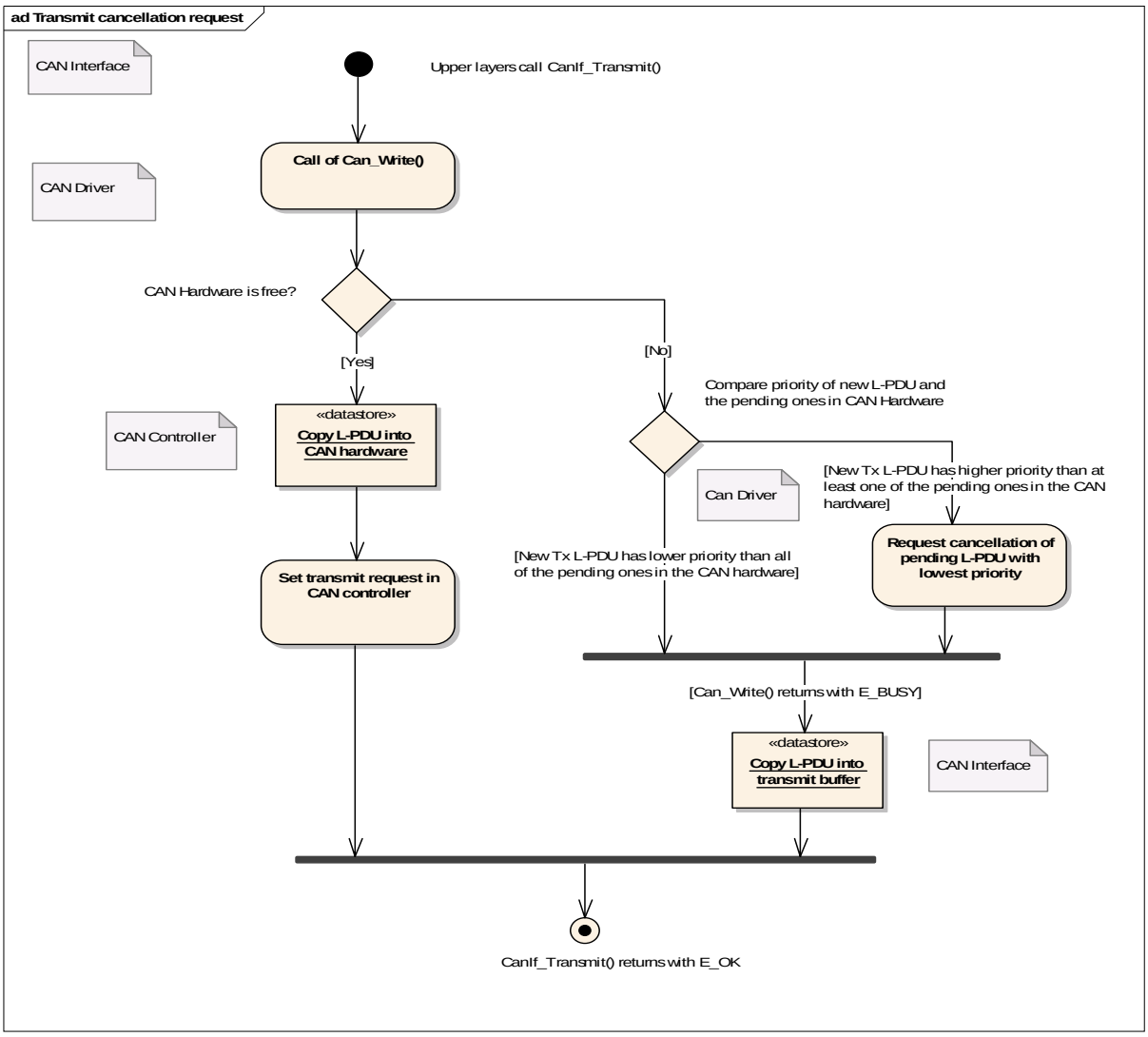

Figure 10 Transmit cancellation request

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

CANIF177: Transmit cancellation can be enabled and disabled by configuration. This feature can only be activated, as far as transmit buffers enabled. At configuration time is must be prevented, that transmit cancellation can be enabled, whenever transmit buffer configuration is disabled.

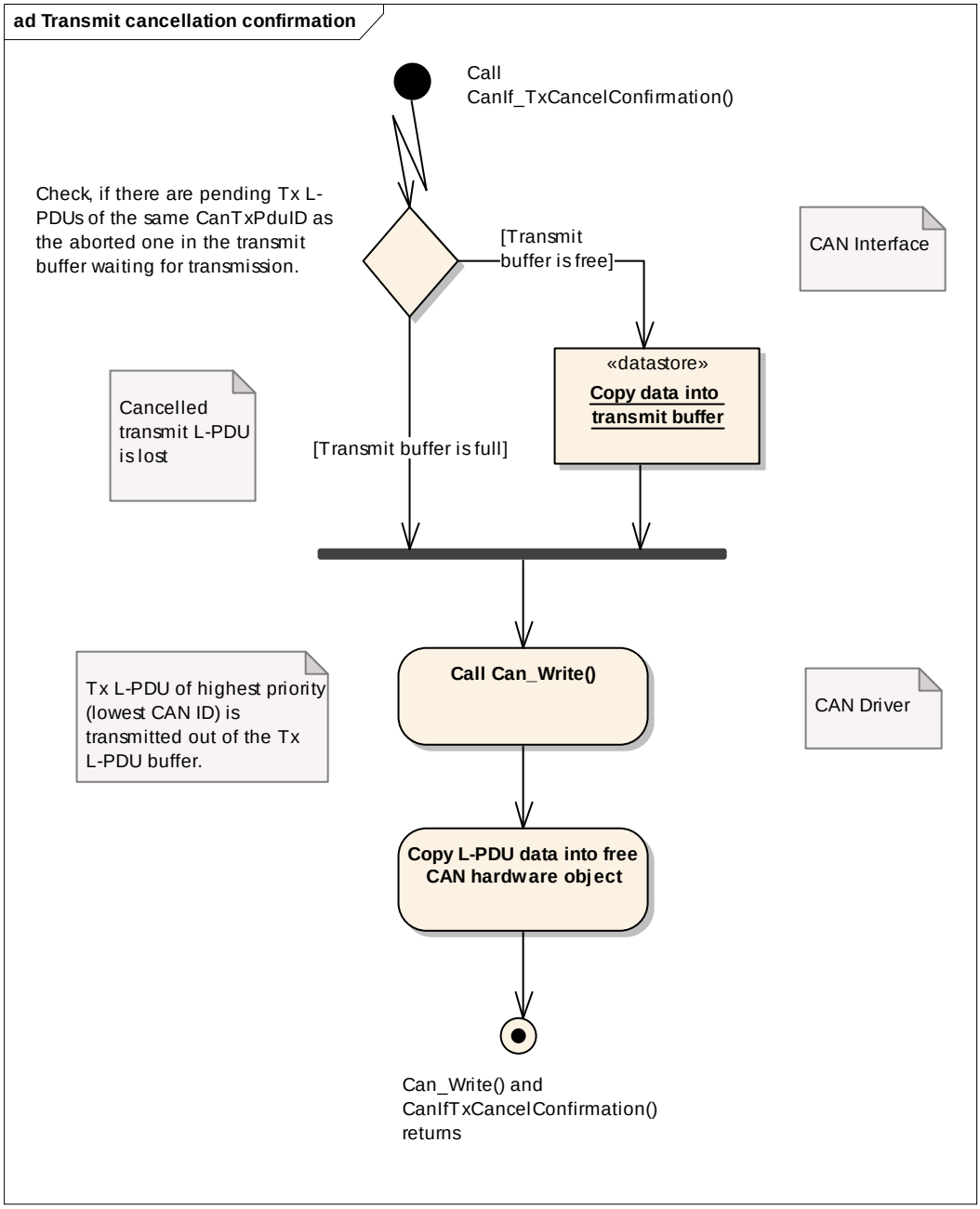

Figure 11 Transmit cancellation confirmation

7.14 Receive data flow

7.14.1 Location of PDU data buffers

CANIF057: According to the AUTOSAR Basic Software Architecture the SDU data buffers are placed in the upper layer communication stacks, i.e. AUTOSAR COM, CAN NM, CAN TP, DCM), where the corresponding data will be evaluated and processed. This means, all transmit as well as all receive SDU buffers are located in these upper layers.

7.14.2 Receive data flow

CANIF134: The usage of the hardware object handle as a parameter in the receive indication call-out service `CanIf_RxIndication()`impacts the design of receive data flow in the CAN Interface. The received data is hardware dependent (nibble and byte ordering, access type) and allocated to the lowest layer in the communication system – to the CAN Driver.
The hardware object handle serves as a link between the CAN Driver and the data customer in the upper layer. The hardware object handle identifies one memory hardware object, where a new CAN L-PDU was received. The target upper layer memory buffer location is derived from the L-PDU Handle, when corresponding L-PDU passed the software filtering, the L-PDU handle was identified and the DLC Check was successfully carried out. In this way the hardware object handle from one side and the L-PDU Handle from another provide a source and destination information for the copying session.

CANIF098: Initially after detection of a reception event the CAN Driver stores the incoming data in an own temporary buffer. If a separate L-SDU normalization is not necessary according to the data structures of the used CAN controller, temporary buffering can be omitted. Thus this feature is up to the CAN Driver. The CAN interface is not able to recognize, whether the CAN Driver uses temporary buffering or a direct hardware access.

CANIF112: The CAN hardware object of the CAN mailbox is locked (refer to CANIF065) until the end of copy process to the temporary or upper layer buffer. The hardware object will be immediately released after the receive indication call-out of the CAN Interface returns to avoid loss of data.

CANIF135: In case temporary buffering is used the hardware object remains locked until the data is read out and copied to the temporary buffer. Then the CAN controller is able to perform the next occurred receive event. The pointer to this temporary buffered L-SDU is used as a parameter in the call-out service `<User_Rx Indication>()`. In this way the pointer on the received L-SDU reaches the data customer in the upper layer. The indication service delivers the target L-PDU handle as parameter. The destination memory buffer is reconstructed from the target L-PDU handle and the communication layer starts data copy. The temporary buffer with the currently received L-SDU is locked all time until the end of copying. After return of CAN Interface's indication services the CAN Driver is responsible for unlocking.

CANIF146: In case no temporary buffer is used the hardware object remains locked until the data is read out and the indication service returns. In this case the parameter of the receive indication call-out refers to the locked CAN RAM with received data.

CANIF147: Both underlying components, CAN Driver and CAN Interface, access the same temporary intermediate buffer, either the CAN RAM or the temporary buffer in the CAN Driver. The CAN Driver may update the L-SDUs and the CAN Interface service `CanIf_RxIndication()` is used to pass access location of the received data to the upper layer. Before this notification is called, the CAN Driver is responsible, that this temporary intermediate buffer (refer to [9.7 Receive indication (interrupt mode)] and [9.8 Receive indication (polling mode)]) is locked.

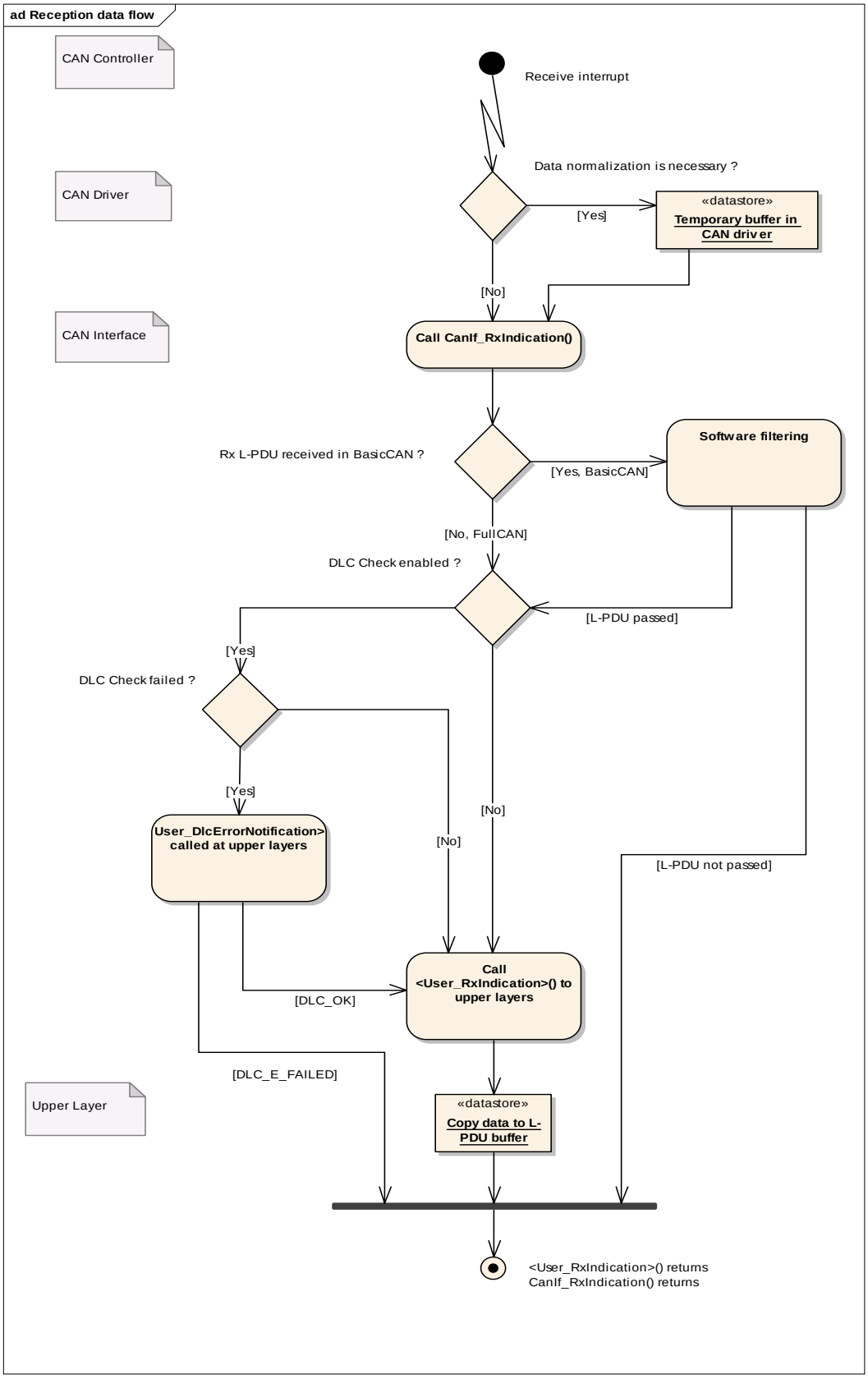

Figure 12 Receive data flow

7.15 Receive indication

CANIF212: After successful reception of a new CAN L-PDU, which passed the hardware acceptance filtering, it is evaluated for acceptance and prepared for later access by the upper communication layers (CANIF012). Upper layers are notified about this asynchronous event, if this CAN L-PDU is successfully detected and accepted for further processing.
During the reception validation first of all the CAN ID of the received L-PDU is compared with L-PDU IDs assigned for further processing in the local ECU. This treatment is called Software Filtering and takes only place for BasicCAN reception (refer to [7.7 BasicCAN and FullCAN reception]).
Afterwards DLC check is processed, if it was enabled during configuration. The DLC Check compares the data length of the received L-PDU with the predefined referenced data length. For further details please refer to chapter [7.22 DLC check].
If the L-PDU passes all validation mechanisms, it is dispatched and the corresponding upper layer is notified. After notification the upper layer may access the data in order to copy them into its own specified memory buffer.

CANIF055: In the event triggered approach all above described operations will be initiated on receive interrupt level. If the basic software runs in polling mode the CAN Driver polling process is responsible for recognizing of new events like reception. After detection of a new reception event the CAN Interface takes the control over the subsequent processing by call of the service `CanIf_RxIndication()`. In dependence to the used notification method (interrupt/polling), the call context of the receive indication can differ: the receive interrupt service itself or an activated task, which calls the corresponding services.

CANIF159: The mentioned note CANIF053 in chapter [7.11 Transmit confirmation] is significant for the receive indication processing too.

On L-PDU reception the CAN Interface Layer performs:
- Software Filtering (only BasicCAN),
- DLC check (optionally),
- Receive data dispatch and
- Protocol dispatching for upper layer receive indication.

CANIF056: The upper layers are notified by `<User_RxIndication>()`. Depending on the different needs of provided information (i.e. AUTOSAR COM, CAN TP) the provided parameters of the corresponding API call-out services can differ. Therefore the CAN Interface can handle several different types of indication call-out services. Please refer to [8.5 Expected interfaces].

CANIF110: A single receive L-PDU can only be assigned to a single indication call-out service.

7.16 Read received data

CANIF196: The read received data API (CANIF194) is a common interface for upper layers to read CAN L-PDUs previously received from the CAN network. The upper communication layers initiate the receive request only via the CAN Interface services

without direct access to the CAN Driver. The initiated receive request is successfully completed, if the CAN Interface write the received CAN L-PDU into the upper layer buffer.

CANIF197: The API service `CanIf_ReadRxPduData()` makes possible to read out data without dependence of reception event. When it is enabled at configuration, necessarily no receive indication service for the same CAN L-PDU for data copy has to be configured. If this indication notification is needed by the upper layer, it can be enabled, too.
By this way the type of mechanism to receive CAN L-PDUs can be chosen at configuration time according to the needs of the upper layer, where the corresponding receive CAN L-PDU belongs to. For details please refer to [9.9 Read received data].

CANIF198: Inside the CAN Interface a single static receive buffer is necessary for each received CAN L-PDU. This buffer is reserved, if the receive request API is enabled at pre-compile time configuration and the corresponding Rx buffer is enabled for the receive L-PDU.

CANIF199: This buffer is filled after the L-PDU is successfully received after e.g. passing the software filtering. During the call of `CanIf_ReadRxPduData()` the receive buffer for BasicCAN L-PDUs is internally locked for access by the calling upper layer.

7.17 Read notification status

CANIF200: In addition to the notification call-outs the API service `CanIf_ReadTxNotifStatus()` is provided to read the transmit confirmation status of any transmit CAN L-PDU. The API service `CanIf_ReadRxNotifStatus()` is provided to read the receive indication status of any receive CAN L-PDU.

CANIF203: This API service can be enabled/disabled globally or per CAN L-PDU at pre-compile time configuration.

CANIF204: If this API service is enabled, the CAN Interface sets the notification status inside during call of `CanIf_TxConfirmation()` or `CanIf_Rx Indication()`for all transmit and receive CAN L-PDUs.

CANIF205: The notification status is cleared after it is read by upper layer. That means: when `CanIf_Tx/RxReadNotifStatus()`is called, the CAN L-PDU notification status inside the CAN Interface is reset. 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

CANIF058: The CAN software stack determines which data coherency and consistency strategy has to be used. The CAN Interface provides an automatic data integrity mechanism, in which a read always returns the value written by the most recent write. An attempt to update the data in the user memory buffers as well as in the internal CAN Interface buffers shall be done with respect on possible changes done in the context of an interrupt service routine or other preemptive events. Preemptive events probably occur either from preemptive tasks, multiple CAN interrupts, if multiple physical channels i.e. for gateways are used, or in case of other peripherals or network systems interrupts, which have the needs to transmit and receive CAN L-PDUs on the network.
Therefore the CAN Interface needs to guarantee internal data integrity.
For this purpose i.e. CAN controller interrupt locks may be used. If polling mode is used also there can be a preempting process treatment that may interrupt the subsequent running.

CANIF064: Handling of shared transmit and receive L-PDU buffers are critical issues for the implementation of the CAN Interface. Therefore the CAN Interface must ensure data integrity and thus use appropriate mechanisms for access to shared resources like transmission/reception L-PDU buffers. Preemptive events, i.e. transmission and reception event from other CAN controllers must be protected against each other.

CANIF065: The CAN Interface may i.e. use the CAN Driver services to enable `(Can_EnableControllerInterrupts())`and disable `(Can_Disable-ControllerInterrupts())` 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. The copying of data, the change of static variables, counters and semaphores shall thus be carried out in these critical sections. It is up to the implementation to use appropriate mechanisms to guarantee data integrity, interruptability and reentrancy.

CANIF077: The transmit request API `CanIf_Transmit()` must be able to operate re-entrant to allow multiple transmit request calls caused by different preemptive events of different L-PDU Handles. The CAN Driver transmit request API `Can_Write()`operates re-entrant as well.

7.19 CAN Controller mode

7.19.1 General functionality

CANIF059: The CAN Interface provides services for controlling the communication mode of all supported CAN controllers represented by the underlying CAN Drivers.

That means that all CAN controllers are controlled by the corresponding provided API services to request and read the current controller mode.

Like the other CAN Driver services even the CAN controller status information are accessible only via the CAN Interface services.
The CAN controller status may be changed at request of the upper layer by the calling of `CanIf_SetControllerMode()` service. The request is validated and passed by the CAN Interface by the CAN Driver API to the addressed CAN controller
The consistent management of all CAN controller connected at one CAN network us the task of the CAN State Manager. By this way the CAN State Manager is responsible to set all CAN controllers of one CAN network sequentially to sleep mode or to wake them up.
When the CAN controller signals a network event (BusOff, wake up), the responsible upper layers (CanSM, EcuM) are notified, that belongs to this service defined during configuration. The CAN Interface provides notification of BusOff, wakeup events and validated wakeup events by means of call-out services. These notifications are performed by the call-out services `<User_ControllerBusOff>()`, `<User_SetWakeupEvent>()` and `<User_ValidationWakeupEvent>()`. In this way the EcuM as well as the CanSM are able to control the system behavior concerning the BusOff recovery or wake up procedure.

7.19.2 CAN Controller operation modes

The CAN Interface stores the operation mode of the CAN controllers according to successful mode transitions after `Can_SetControllerMode()`. According to the requested operation mode by the CAN State Manager the CAN Interface translates them into the right order of mode transitions for the CAN controller.

CANIF081: The needed network related state machine is implemented in the CAN State Manager. Refer to [11] Specification of CAN State Manager. The CAN Interface only stores the requested mode and executes the requested transition.

7.19.2.1 CANIF_CS_UNINIT

CANIF213: The CAN Interface is not initialized. The Ecu State Manager (EcuM) has to consider, that also the CAN driver(s) and CAN Controller(s) shall also not be initialized.

7.19.2.2 CANIF_CS_STOPPED

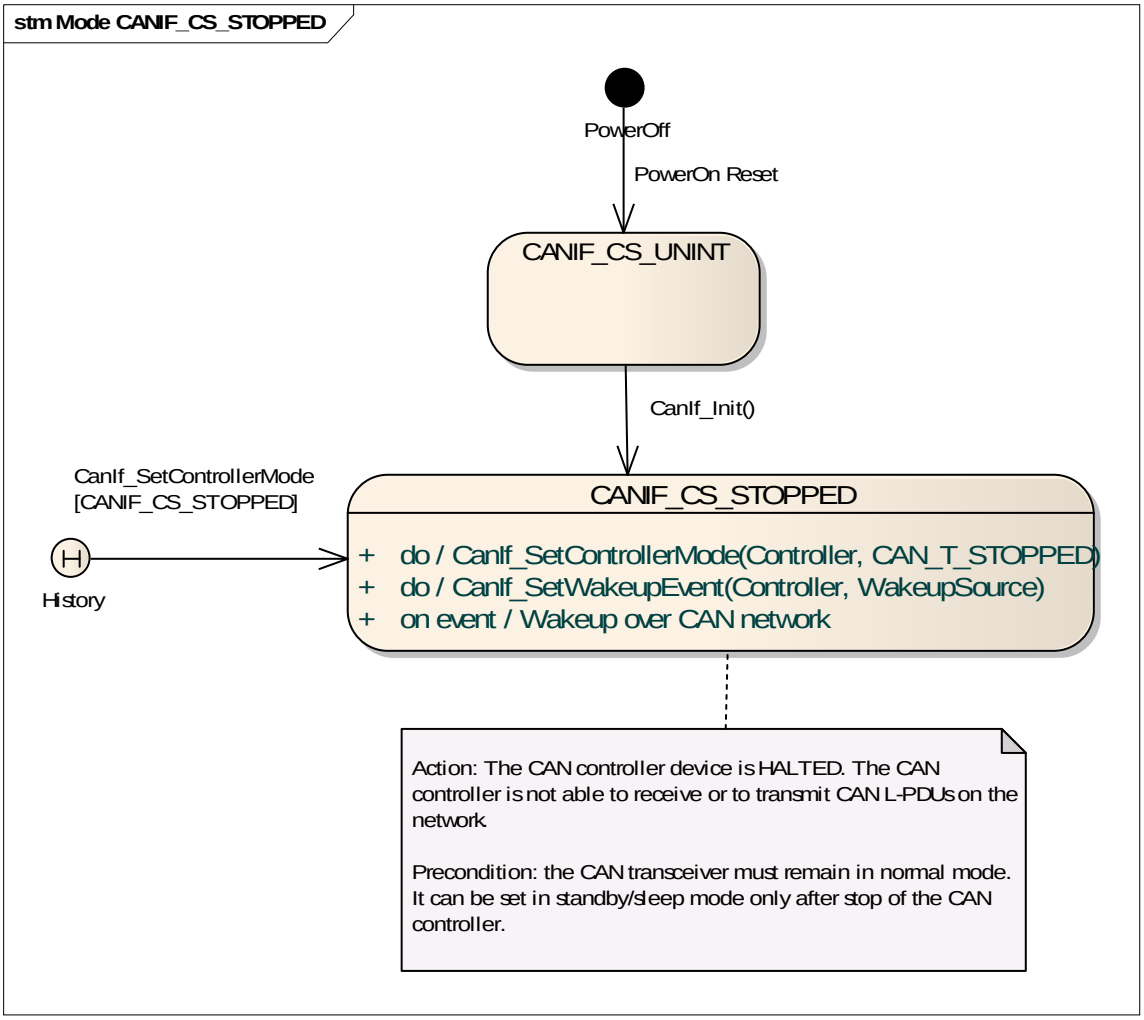

Figure 13 Activities of STOPPED mode transition

CANIF214: The CAN Interface suppresses subsequent transmit requests to the CAN Driver because the CAN controller shall be prevented from sending CAN L-PDUs. If the used CAN controller provides a STOPPED mode, it will be used. `CanIf_Transmit()` returns `E_NOT_OK`. The PDU Mode is set to CANIF_OFFLINE (refer to [7.20 PDU channel mode control]). All pending transmit requests are canceled. All contents of transmit buffers are deleted. In the mode CANIF_CS_STOPPED no CAN L-PDUs can be received.

CANIF298: STOPPED Mode is entered automatically for that CAN controller, where a BusOff event has been signaled by the corresponding CAN Driver.

The paragraph below reflects 4 main important uses cases described below for transition to CANIF_CS_STOPPED mode. Concerning each use case the occurred event at the CAN Interface (call of it's API) and the subsequent required action is explained:
Use Case ***Initialization***:
Event: `CanIf_Init()`
Action: Internal initialization of the CAN Interface.

Use Case ***STOP Network***:
Event: `CanIf_SetControllerMode(Controller, CANIF_CS_STOPPED)`
Action: `Can_SetControllerMode(Controller, CAN_T_STOPPED)`

Use Case ***WAKEUP Network***:
Event: `CanIf_SetControllerMode(Controller, CANIF_CS_STARTED)`
Action: `Can_SetControllerMode(Controller, CAN_T_WAKEUP).`

Use Case ***BusOFF***:
Event: CAN controller went BusOFF
Action: CAN Driver ensures, that CAN controller is STOPPED

7.19.2.3 CANIF_CS_STARTED

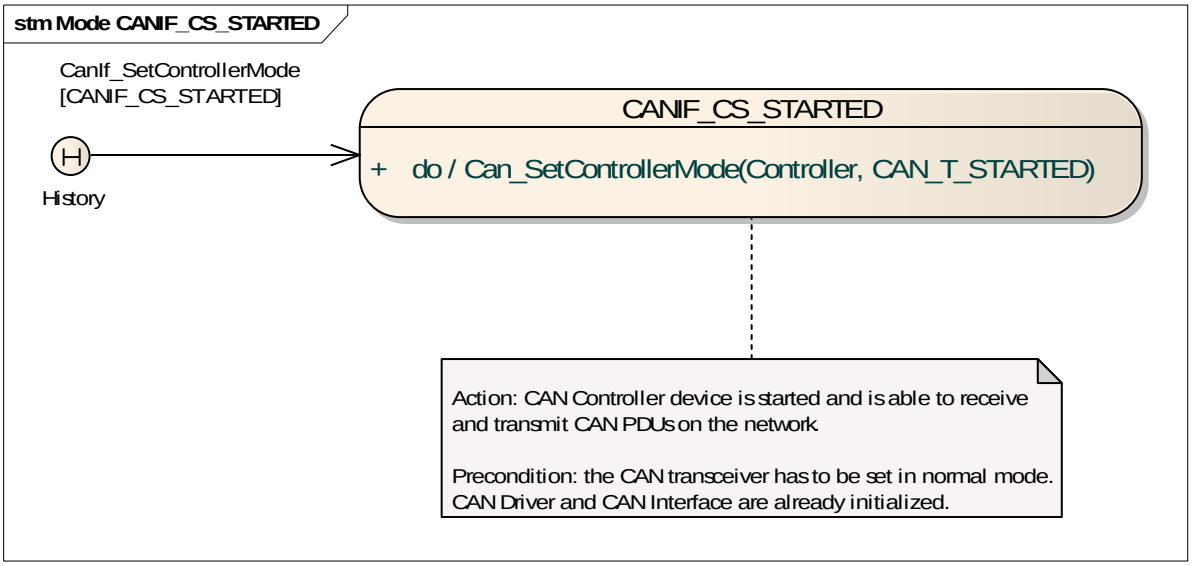

Figure 14 Activities of STARTED mode transition

CANIF215: The controller is fully operational at the CAN network; all transmit requests thus are passed to the CAN Driver. CAN L-PDUs can be received and are notified to upper layers. The PDU mode is set to CANIF_ONLINE.
To start a network following API call event and subsequent action of the CAN Interface is required to perform the transition to CANIF_CS_STARTED mode:

Use Case ***START Network***:
Event: `CanIf_SetControllerMode(Controller, CANIF_CS_STARTED)`
Action: `Can_SetControllerMode(Controller, CAN_T_STARTED)`

7.19.2.4 CANIF_CS_SLEEP

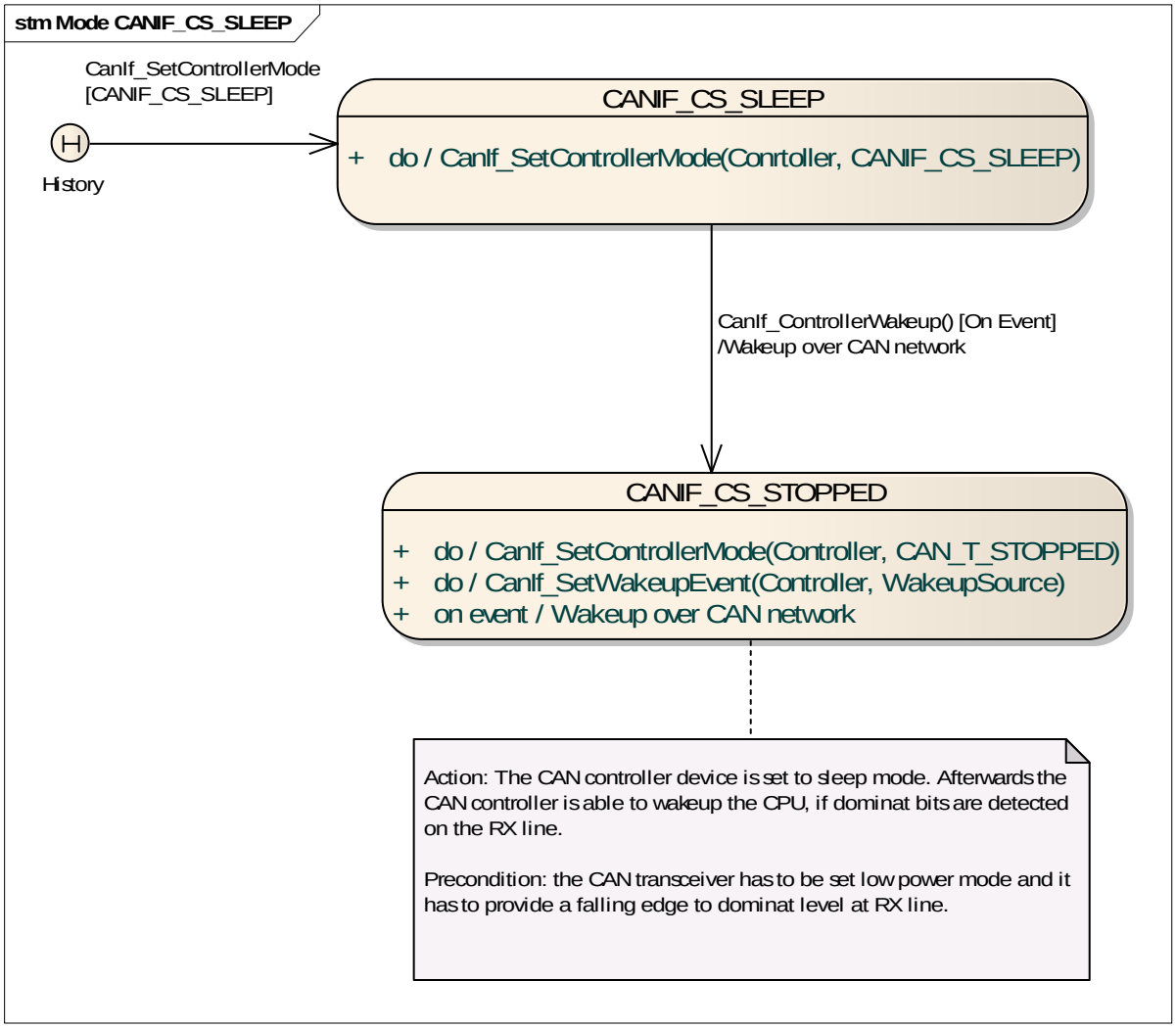

Figure 15 Activities of SLEEP transition

CANIF216: The CAN controller is set to SLEEP mode and its own wakeup interrupts are enabled, if supported. As long as wakeup functionality is not provided by the CAN controller, the CAN Driver encapsulates it.
To set network to SLEEP following API call event and subsequent action of the CAN Interface is required to perform the transition to CANIF_CS_SLEEP mode:

Use Case ***SLEEP Network***:
Event: `CanIf_SetControllerMode(Controller, CANIF_CS_SLEEP)`
Action: `Can_SetControllerMode(Controller, CAN_T_SLEEP).`

7.19.2.5 BUSOFF

BusOff is a transition from STARTED to STOPPED mode.

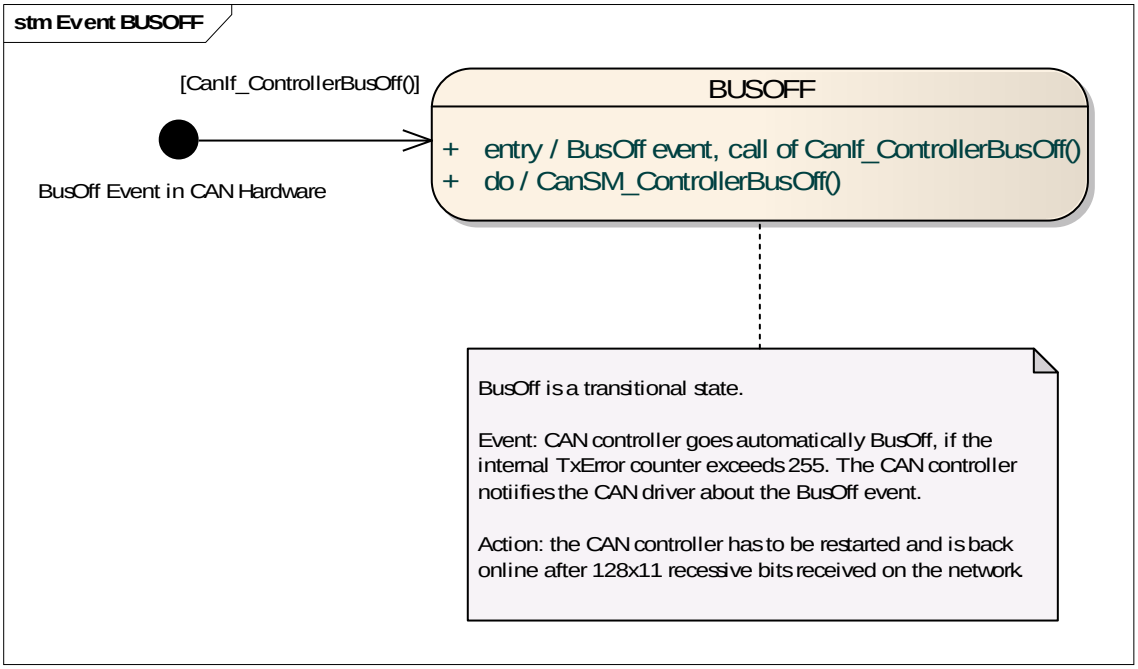

Figure 16 Activities of BUSOFF transition

7.19.3 Controller mode transitions

CANIF078: The API for state change requests to the CAN controller behaves in a synchronous manner without any asynchronous notification via call-out services. The real transition to the requested mode occurs asynchronously based on setting of transition requests in the CAN controller hardware, i.e. request for sleep transition `CANIF_CS_SLEEP`. After successful change to SLEEP mode the CAN Driver service `Can_SetControllerMode()` and as well the CAN Interface `CanIf_SetControllerMode()`returns with `E_OK`. In case of an unsuccessful mode transition the CAN Interface returns `E_NOT_OK`. Mode transitions `CANIF_CS_STARTED` and `CANIF_CS_STOPPED` are treated synchronously as well. This synchronous behavior makes it possible for i.e. the CAN State Manager to handle the mode transitions of the CAN networks.

CANIF093: The current CAN Interface operation mode can be polled from upper layers by `CanIf_GetControllerMode()`.

CANIF079: Sleep and Wakeup mode is not supported by all types of CAN controllers.These modes are encapsulated by the CAN Driver by providing hardware independent operation modes over its interface, which has to be managed by the CAN Interface. Whereas the transitions to STARTED and STOPPED mode returns synchronously without subsequent check, only during the request to sleep transition the CAN Driver checks the underlying CAN controller, whether the sleep request was executed successfully or not. In this case transition request returns immediately, whereas the sleep transition of the CAN controller is delayed. The transition to sleep mode is aborted after a fixed time the transition was not successful. The CAN Driver

may release directly a wakeup interrupt during the transition request, when CAN L-PDUs are transmitted or received at the same time.

This treatment guarantees, that the CAN State Manager is informed immediately about the transition to SLEEP mode for handling the CAN Transceiver and enabling the wakeup interrupt.

CANIF080: After transition to STARTED mode `CANIF_CS_STARTED` all transmission and reception events are processed. After request of STOPPED mode `CANIF_CS_STOPPED` the CAN Interface suppresses all following transmission requests and no reception events are further processed.

CANIF089: The CAN Interface distinguishes between internal initiated CAN controller wake up request (internal request) and network wake up request (external request). The first one is an internal synchronous request; the second is a CAN controller event. Only network initiated wakeups are notified by the wakeup notification as far as it is supported by the used CAN controllers.

CANIF090: If the physical ECU belongs to multiple networks, each CAN network must be controlled by its own station management (CAN State Manager) and/or network management (CAN NM).

7.19.4 Wakeup and validated wakeup events

CANIF227: If the ECU shall support wakeup over CAN network, regardless of the used wakeup method (directly about CAN controller or CAN transceiver), the CAN Driver and the corresponding CAN controller has to be set to SLEEP mode. Only this mode ensures that the CAN controller is stopped, thus the wakeup interrupt can be enabled.

CANIF180: The CAN Interface supports wakeup notification only, if

- ⇨ underlying CAN controller provide wakeup support and wakeup is enabled by CAN Driver configuration.
- ⇨ underlying CAN transceiver provide wakeup support and wakeup is enabled by CAN Transceiver Driver configuration.
- ⇨ the CAN Interface is in the mode CANIF_CS_SLEEP.

CANIF181: If wakeup support is enabled, CAN Interface is notified by the ECU firmware about a general CAN wakeup event by the call-out service `CanIf_CheckWakeup()`. Is API is invoked in interrupt as well as in polling mode.

CANIF285: The CAN Interface queries at all CAN Drivers or CAN Transceivers according to the configuration, which exact CAN hardware device caused the wakeup event over CAN.
The corresponding device driver returns to the CAN Interface and provides the requested wakeup device information. For details see respective UML diagram in the chapter “CAN Wakeup Sequences” of document [15] Specification of ECU State Manager.

CANIF231: After notification of a wakeup event the CAN Interface shall not start the CAN controller automatically. The CAN State Manager must set the CAN Interface from STOPPED into STARTED mode for the corresponding CAN controller. This is required, because wakeup validation can take place only in STARTED mode to be able to detect the first received CAN message after a wakeup event.

CANIF232: After a wakeup event occurred, the CAN Interface stores this wakeup event until the wakeup event is validated. During validation of the wakeup event and also during re-initialization of the CAN Interface all temporarily stored wakeup events are reset.
Attention: the CAN Interface notifies the upper layers about received messages after transition to CANIF_CS_STARTED. The CAN Interface does not wait for processing notifications until a wakeup event was successfully validated.

CANIF226: Validation wakeup only takes place, if
- ⇨ wakeup support is enabled.
- ⇨ the CAN Interface is in the mode CANIF_CS_STARTED.

CANIF286: After CAN Interface has been notified by an wakeup event, it enables the detection for CAN wakeup validation. Therefore the CAN Interface stores a successful validation, whenever the indication callback is called by the CAN Driver to notify about the first CAN L-PDU successful received after the wakeup event occurred.

CANIF182: The CAN Interface notifies the ECU State Manager about a validated wakeup event by call of `<User_ValidationWakeupEvent>()` during call of `CanIf_CheckValidation()`. For details see respective UML diagram in the chapter “CAN Wakeup Sequences” of document [15] Specification of ECU State Manager.

7.20 PDU channel mode control

7.20.1 PDU channel groups

CANIF060: Each L-PDU is assigned to one dedicated physical CAN channel connected to one CAN controller and one CAN network. By this way all L-PDUs belonging to one physical channel can be controlled on the view of handling logically single L-PDU channel groups. Those logical groups represent all L-PDUs of one ECU connected to one underlying CAN network.
The figure below shows one possible usage of L-PDU channel group and its relation to the upper layers and/or networks:

CANIF088: An L-PDU can only be assigned to one channel group.

CANIF217: Typical users like PDU Router or the network management are responsible for controlling the PDU operation modes.

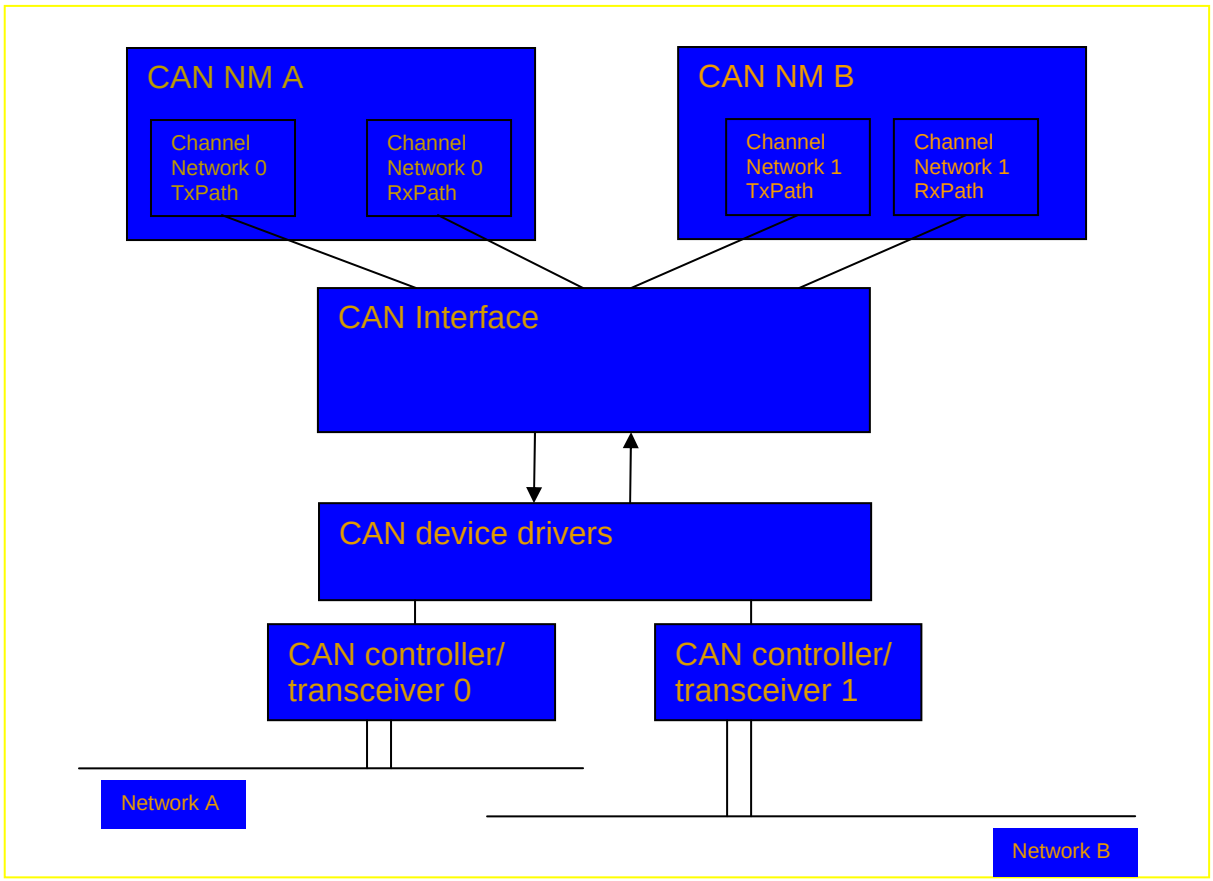

Figure 17 Channel L-PDU groups

7.20.2 PDU channel modes

CANIF027: The CAN Interface provides services to prevent the processing of
- all transmit L-PDUs of the own ECU belonging to one logical channel,
- all receive L-PDUs of the own ECU belonging to one logical channel,
- all transmit and receive L-PDUs of the own ECU belonging to one logical channel
- even all L-PDUs.

Every PDU mode change can be requested for transmission and reception path separately or commonly. A change of the channel mode has only an effect during the network mode `CANIF_CS_STARTED` (refer to [0:]). The change of the channel mode is performed but in STOP, SLEEP or UNINIT state no L-PDUs are transmitted nor received since the CAN controller is not in STARTED mode**.**

7.20.2.1 OFFLINE Mode

CANIF073: In offline mode all L-PDUs of the corresponding channel are prevented for transmission and reception. Thus at transmission no transmit requests are passed to the CAN Driver and no transmit confirmation notifications are processed. At reception no receive indications to upper layers are executed. The transmit buffers

are cleared. Transmit requests return `E_NOT_OK`. The transmit path as well the receive path is offline. This is the default channel mode after initialization.

CANIF118: The BusOff notification is automatically and thus implicitly suppressed due to in OFFLINE mode no L-PDUs can be transmitted and thus the CAN controller is not able to go in BusOff mode (CAN specification). If pending L-PDUs in the CAN hardware are transmitted after change to OFFLINE mode and BusOff occurs, the BusOff notification is not suppressed. The wakeup notification is not affected concerning mode changes between ONLINE/OFFLINE.

7.20.2.2 ONLINE Mode

CANIF074: Probably appearing confirmations from previous transmit requests to the CAN Driver released in Tx online mode are not suppressed by the CAN Interface.

CANIF075: The online mode enables the reception and/or the transmission path to the CAN Driver. It re-activates the assignment of transmit and receive PDUs to a defined physical channel. The online mode cancels the lock made by offline mode call. Every change back to offline mode clears the transmit buffers. The appropriate CAN Interface service is called `CanIf_SetPduMode()`.

7.20.2.3 ONLINE/OFFLINE Mode for Tx/Rx path

CANIF096: The Tx/Rx online mode and the Tx/Rx offline mode only offers the possibility to change the channel mode on the Rx/Tx paths separately. This modes behave the same like online/offline, but only for the transmit L-PDUs or the receive L-PDUs of the corresponding channel.

CANIF095: The CAN Interface provides information about the status of 'online'/'offline' service when required via the service `CanIf_GetPduMode()`.

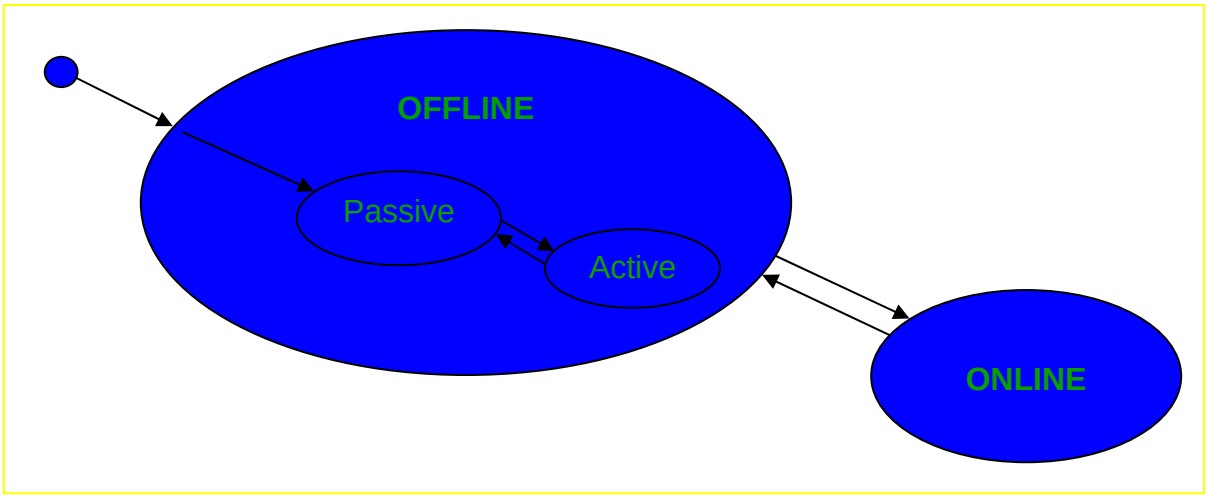

Figure 18 PDU channel mode control

7.20.2.4 OFFLINE ACTIVE Mode

CANIF072: The CAN Interface provides simulation of successful transmission by the offline active mode. This mode only affects the transmission path. By this mode confirmation handling is performed synchronously at the end of the transmit request, but no transmit request is passed to the CAN Driver. On logical view the offline active mode is a sub-mode of the offline mode, whereas it can be enabled in online as well as in offline mode.
The offline active mode is enabled by call of `CanIf_SetPduMode (CANIF_SET_TX_OFFLINE_ACTIVE)`. This mode can be left by `CANIF_SET_TX_ONLINE` or `CANIF_SET_TX_OFFLINE`.

This functionality is useful realizing special operating modes (i.e. diagnosis passive mode) to avoid bus traffic without impact to the notification mechanism and thus is typically used for diagnostic usage.
Note: During the Tx Offline Active mode the upper layer has to handle the execution of the confirmations. The confirmation handling is executed immediately at the end of the transmit request.

The figure above shows a diagram with possible L-PDU channel modes. Each L-PDU channel can be offline (no transmission) or online (activated transmission). A simulation of the successful sending (transmit confirmation) is supported in the offline mode and called offline active mode. The default state of L-PDU channel in offline mode thus is passive. No simulation of the successful transmission takes place.

7.21 Software receive filter

CANIF025: 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. These L-PDUs are filtered out and further software processing is prohibited.

CANIF094: Certain software filter algorithms are provided to optimize software filter runtime. The approach of software filter mechanisms is to find out the corresponding L-PDU handle from the HRH and CAN ID currently being processed. After the L-PDU handle is found it enables upper layers to access L-PDU information directly.

7.21.1 Software filtering concept

CANIF234: 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).

CANIF237: For definition of range reception it is necessary to define at least one Rx L-PDU with the CAN Id inside the defined range. The range is defined by its upper and lower limit CAN Id.

CANIF030: 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 CAN Identifiers 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 CAN Identifiers by comparing them with the new received CAN Identifier. In case of a hit the receive L-PDU handle is derived from the found CAN Identifier.

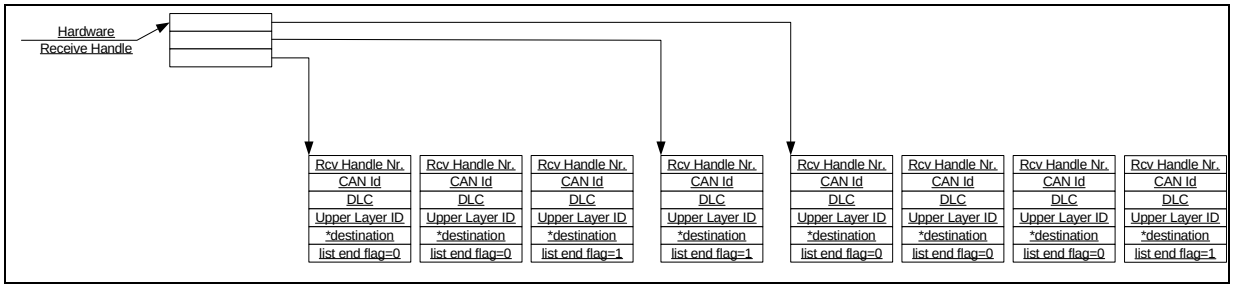

Figure 19 Software filtering example

7.21.2 Software filter algorithms

CANIF097: The choice of suitable software search algorithms it is up to the implementation. According to the wide range of possible receive BasicCAN operations 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

CANIF026: The DLC of the received L-PDU is compared with the expected, statically configured DLC for the received L-PDU. The statically defined DLC value shall be derived from the size of used bytes inside this L-PDU. The 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. All CAN L-PDUs with a DLC equal or greater then the expected DLC will be accepted.

CANIF297: The number of bytes that are later on copied corresponds to the expected DLC of the received L-PDU, not to the current received DLC.

CANIF296: All upper protocol layers, which operate with a dynamic length and managing the own data buffer for receive data, the DLC check shall be disabled by setting DLC to NULL. In that case the DLC check always passes.

CANIF262: Also in case of DLC check is disabled, the data length passed the indication notification to the upper layers, which copies the data to its own buffer always shall correspond to the current received DLC value and not to the predefined expected DLC. By this way always the current received data are copied although the DLC check is disabled by setting the expected DLC value to NULL.

CANIF166: The expected DLC may be defined for one specific L-PDU or a group of PDUs.

CANIF168: If the DLC check fails, only the DEM shall be notified. Other upper layers are not informed. No receive indication is executed.

CANIF031: The DLC Check shall be enabled or disabled globally by CAN Interface configuration for all used CAN Driver.

7.23 L-PDU dispatcher to upper layers

CANIF024: Upper Communication Layers use the unified interface of the CAN Interface for transmission and reception.
At reception side each L-PDU handle belongs to one single upper layer as destination for the corresponding receive L-PDU or group of such L-PDUs. This relation is assigned statically at configuration time. The task of the L-PDU dispatcher inside of the CAN Interface is to find out the customer for a received L-PDU and to dispatch the indications towards the found upper layer.
At transmission side the L-PDU dispatcher has to find out the corresponding Tx confirmation call-out service of the target upper layer.
Receive Indication as well as transmit confirmation notifications are processed via the corresponding call-out services. These call-out services may exist several times with different names defined in the notified upper layer modules. Thus every upper layer module receiving CAN L-PDUs can be notified by its own indication service, for example `<User_RxIndication>()`.These services are statically configured, depending on the layers that have to be served.

7.24 Polling mode

CANIF029: The polling mode provides handling of transmit, receive and error events occurred in the CAN hardware without the usage of hardware interrupts. Thus the CAN Interface and the CAN Driver provides services for detection and execution corresponding hardware events.
The CAN Interface API's characteristic and syntax does not change in polling mode. By this way upper layers are abstracted from the strategy to detect hardware events. If different CAN Drivers are in use, the calling rates shall be harmonized during configuration setup and system integration.

These services are able to detect new events that occurred in the CAN Hardware since its last call. The CAN Interface notification services for forwarding of detected events by the CAN Driver are the same like for interrupt operation:

- Receive service – detects new PDU reception events and calls `CanIf_RxIndication()`.
- Transmit service – controls new confirmations and calls `CanIf_TxConfirmation()`.
- Error service – detects errors occurred during PDU processing (BusOff) and calls `CanIf_ControllerBusOff()`.
- Wakeup service – detects a CAN controller / CAN transceiver wake up event and calls `CanIf_SetWakeupEvent()`.

Please refer to chapter [8.4] for further details to these notification callbacks.

CANIF129: The calling context of the notification call-outs differs between Interrupt and polling mode. Whereas in interrupt mode the notifications are performed on Interrupt level, these are invoked on task level in polling mode.
If any access to the CAN controller's mailbox is blocked, subsequent transmit buffering takes place (refer [7.12 Transmit buffering]).

CANIF130: The Polling and Interrupt mode can be configured for each underlying CAN controller.

7.25 Multiple CAN Driver support

CANIF028: A specific mapping is needed in the CAN Interface to cover multiple CAN Drivers to provide a common interface to upper layers. Thus the CAN Interface must dispatch all actions up-down to the APIs of the corresponding target CAN Driver and underlying CAN controller(s) and as well the way down-up by providing multiple call-out notifications on the CAN Interface for multiple CAN Drivers.

CANIF124: The naming convention is as follows:

`<CAN Driver module name>_<vendorID>_<Vendor specific API name><driver abbreviation>()`
BSW00347 specifies the naming convention.

CANIF224: The naming conventions can be used only in that case, if multiple different CAN controller types on one ECU have to be supported. If only one controller type is used, the original naming conventions without any `<driver abbreviation>` extensions are sufficient.

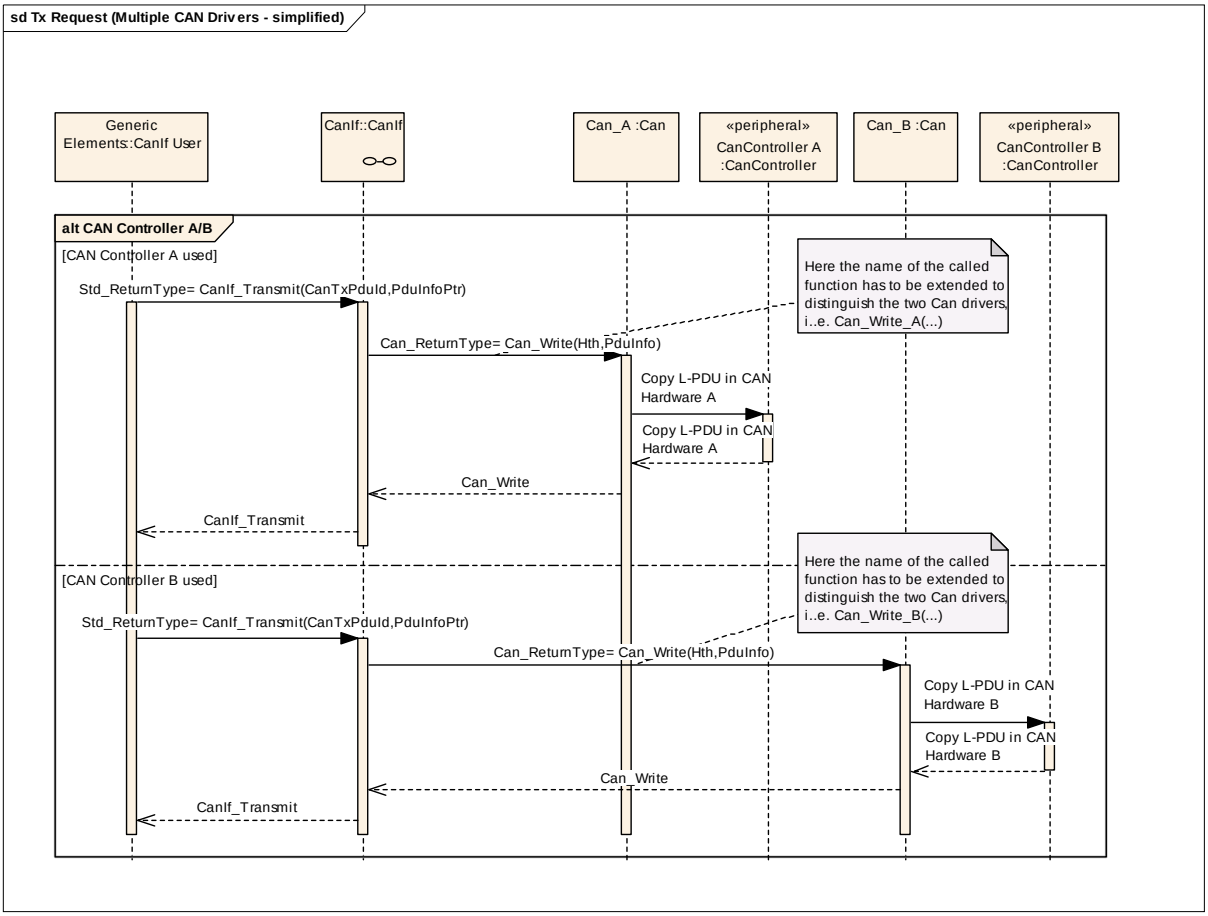

Figure 20 Transmission request with multiple CAN Drivers - simplified

CANIF235: The support for multiple CAN Drivers can be enabled and disabled by the configuration parameter `CANIF_MULTIPLE_DRIVER_SUPPORT`.

7.25.1 Transmit requests by using multiple CAN Drivers

CANIF125: Each transmit L-PDU handle affords deriving the corresponding CAN controller and implicitly the CAN Driver serving the affected hardware unit. Resolving of these dependencies is possible because of the construction of the CAN controller handle: it combines CAN Driver handle and the corresponding CAN controller in the hardware unit.

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

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

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

Example:

Operations called	*Description*
`CanIf_Transmit ( PduId_1, *PduInfoPtr_1 )`	Upper layer initiates a transmit request. The PDU ID 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 group will be extracted from the PduId_1. Each PDU group refers to a network and thus as well the hardware unit number and the CAN controller number. The hardware unit number points on an instance of the CAN Driver 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_A ( Hth, *PduInfoPtr_1 )`	Request for transmission to the CAN_Driver_A serving i.e. CAN controller #1 within the "A" hardware unit.
Hardware request	All L-PDU data will be set in Hardware of i.e. CAN controller #0 within hardware unit "A" and the transmit request enabled.
`CanIf_Transmit ( PduId_2, *PduInfoPtr_2 )`	Upper layer initiates transmit request. The parameter transmit handle leads to another CAN controller and then to another hardware unit. The number of the hardware unit is relevant for the dispatch as it is used as index for the array with pointer to functions. At first the number of the PDU group will be extracted from the PduId_2. Each PDU group refers to a network and thus as well to the hardware unit number and to the CAN controller number. The hardware unit number points on an instance of the CAN Driver 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_B ( Hth, *PduInfoPtr_2 )`	Request for transmission to the CAN_Driver_B serving i.e. CAN controller #1 within the "B" hardware unit.
Hardware request	All L-PDU data will be set in the Hardware of i.e. the CAN controller #1 within hardware unit "B" and the transmit request enabled.

7.25.2 Notification mechanism by using multiple CAN Drivers

CANIF126: Every notification call-out service invoked by the CAN Drivers at the CAN Interface exists multiple times, if multiple CAN Drivers are used in a single ECU. This means, that each used CAN Driver calls 'it's own' call-out service at the CAN Interface. The CAN interface must provide all call-out service unique for each

underlying CAN Driver. Thus the HRH parameter is unique at the scope of each CAN Driver. Following call-out/callback services are affected:

- `CanIf_TxConfirmation`
- `CanIf_RxIndication`
- `CanIf_ControllerBusOff`
- `CanIf_SetWakeupEvent`
- `CanIf_CancelTxConfirmation`

CANIF127: Example: on reception side the corresponding call-out routine of the CAN Driver being triggered by the reception events is called at the CAN Interface. If the CAN Interface underlies two CAN Drivers, two `CanIf_RxIndication()` routines has to be provided. At configuration time the relation between call-out service and used CAN Driver has to be set up.

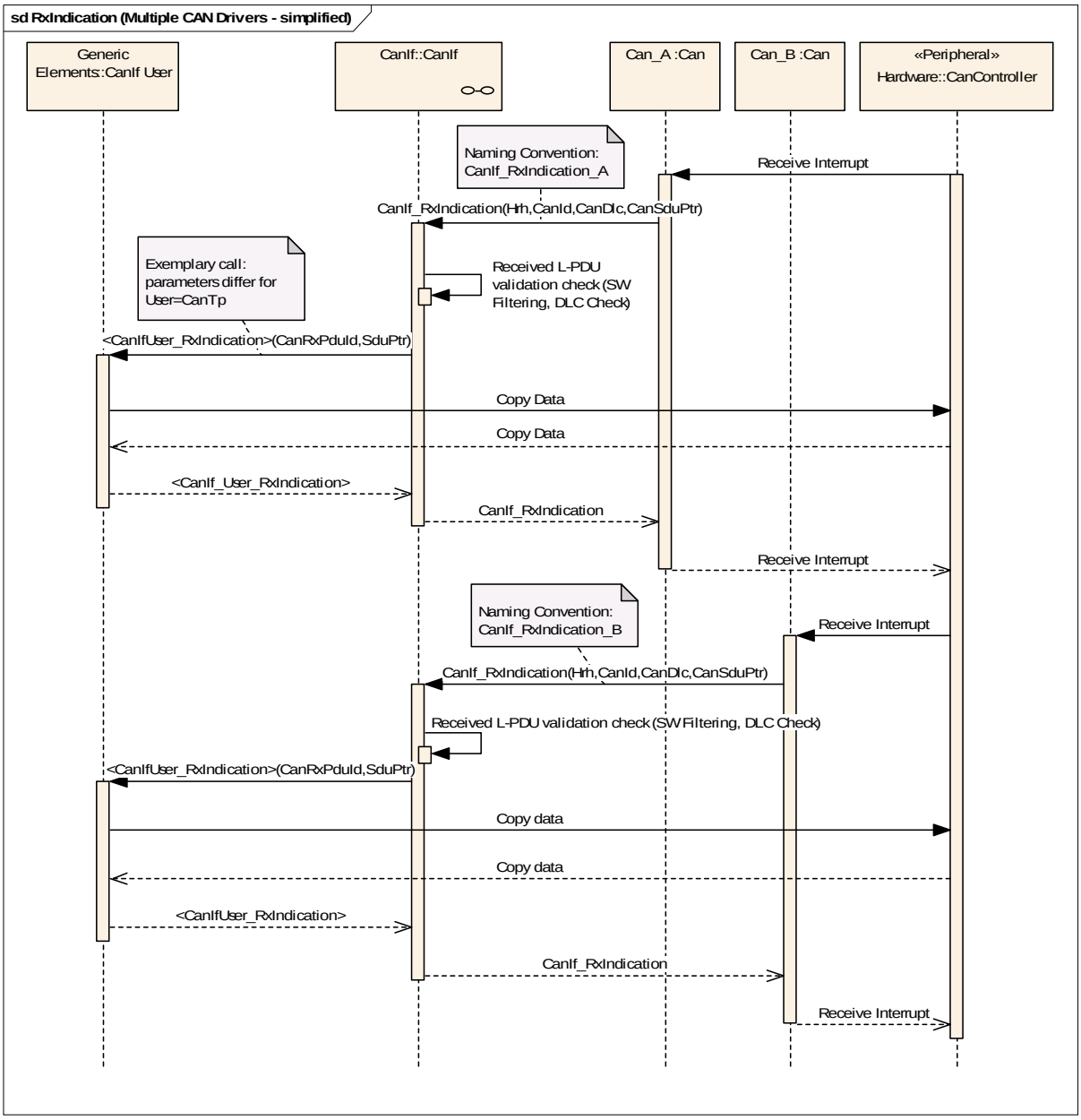

Figure 21 Receive interrupt with multiple CAN Drivers - simplified

Operations called	***Description***
Receive Interrupt	The CAN controller 1 signals a successful reception and triggers a receive interrupt. The ISR of CAN Driver A is invoked.
`CanIf_RxIndication_A (Hrh_3, CanId_1, CanDlc_8, *CanSduPtr_1)`	The reception is indicated to the CAN Interface by calling of `CanIf_RxIndication_A()`. The HRH specifies the CAN RAM hardware object and the corresponding CAN controller (`Hrh_3`), which contains the received L-PDU. The temporary buffer is referenced to the CAN Interface by *CanSduPtr_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-PDU is not indicated to upper layers. Further processing is suppressed. If the L-PDU is found, the DLC of the received L-PDU is compared with the expected, statically configured one for the received L-PDU.

Operations called	*Description*
`<User_RxIndication> (CanRxPduId_4, *CanSduPtr_1)`	The corresponding receive indication service of the upper layer is called. This signals a successful reception to the target upper layer. The parameter CanRxPduId_4 specifies the L-PDU, the second parameter is the reference on the temporary buffer within the L-SDU.
Receive Interrupt	The CAN controller 2 signals a successful reception and triggers a receive interrupt. The ISR of CAN Driver B is invoked.
`CanIf_RxIndication_B (Hrh_3, CanId_5, CanDlc_8, *CanSduPtr_2)`	The reception is indicated to the CAN Interface by calling of `CanIf_RxIndication_B()`. The HRH specifies the CAN RAM hardware object and the corresponding CAN controller (`Hrh_3`), which contains the received L-PDU. The temporary buffer is referenced to the CAN Interface by *CanSduPtr_2.
Validation check (SW Filtering, DLC Check)	The Software Filtering checks, whether the received L-PDU will be processed on a local ECU. If not, the received L-PDU is not indicated to upper layers. Further processing is suppressed. If the L-PDU is found, the DLC of the received L-PDU is compared with the expected, statically configured one for the received L-PDU.
`<User_RxIndication> (CanRxPduId_2, *CanSduPtr_2)`	The corresponding receive indication service of the upper layer is called. This signals a successful reception to the target upper layer. The parameter CanRxPduId_2 specifies the L-PDU, the second parameter is the reference on the temporary buffer within the L-SDU.

7.25.3 Mapping table for multiple CAN Driver handling

CANIF062: A table with addresses to all CAN Driver API services is the basis to provide a unique driver interface to the CAN Interface. This table makes the assignment from two different driver interfaces to one single driver interface (with prefix (Can_).
In case of L-PDU handle based APIs, the corresponding CAN Driver has to be derived from the L-PDU handle. Afterwards the CAN Driver number is used as an index for the table with function pointers. The parameters have correspondingly to be translated: i.e. L-PDU handle => HTH/HRH, CanId, Dlc.

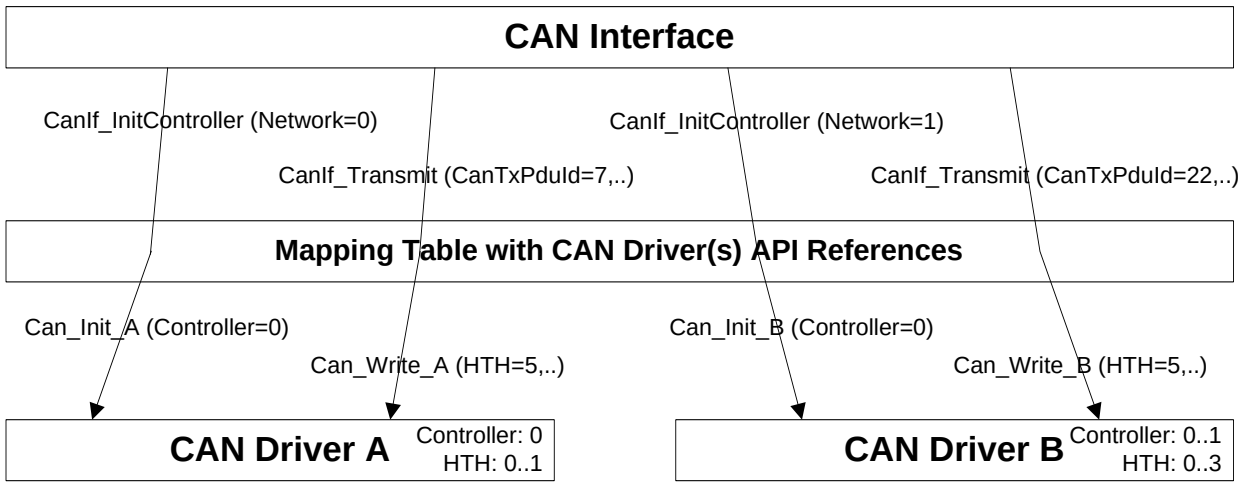

Figure 22 HTH Assignment with multiple CAN Drivers

Each CAN Driver supports a certain number of underlying CAN controllers and a fixed number of HTHs. Each CAN Driver has got an own numbering area, which starts always at 0 for controller and HTH.

7.26 Error classification

CANIF017: This chapter lists and classifies all errors that can be detected within this software module. Each error shall be classified according to relevance (development / production) and related error code. For development errors, a value shall be defined.

CANIF153: Values for production code Event Ids are assigned externally by the configuration of the Dem. They are published in the file `Dem_IntErrId.h` and included via `Dem.h`.

CANIF154: Development error values are of type uint8.

CANIF120: The naming of errors has to be compliant to BSW000327.

CANIF207:

Type or error	*Relevance*	*Related error code*	*Value*

API service called with invalid parameter	Development	`CANIF_E_PARAM_CANID`
		`CANIF_E_PARAM_DLC`
		`CANIF_E_PARAM_HRH`
		`CANIF_E_PARAM_CHANNEL`
		`CANIF_E_PARAM_CONTROLLER`
		`CANIF_E_PARAM_WAKEUPSOURCE`
			`10`
			`11`
			`12`
			`13`
			`14`
			`15`

API service called with invalid pointer	Development	`CANIF_E_PARAM_POINTER`	`20`
API service used without module initialization	Development	`CANIF_E_UNINIT`	`30`
Requested API operation is not supported	Development	`CANIF_E_NOK_NOSUPPORT`	`40`
API service called with invalid transceiver parameter	Production	`CANIF_TRCV_E_TRANSCEIVER`	`Assigned by DEM`
API service called with invalid parameter for CAN transceiver operation mode	Development	`CANIF_TRCV_E_TRCV_NOT_STANDBY`	`60`
API service called with invalid parameter for CAN transceiver operation mode	Development	`CANIF_TRCV_E_TRCV_NOT_NORMAL`	`70`
Transmit PDU ID invalid	Development	`CANIF_E_INVALID_TXPDUID`	`80`
Receive PDU ID invalid	Development	`CANIF_E_INVALID_RXPDUID`	`90`
Failed DLC Check	Production	`CANIF_E_INVALID_DLC`	`Assigned by DEM`
CAN Interface is in STOPPED mode	Production	`CANIF_E_STOPPED`	`Assigned by DEM`
Transmit buffers full	Production	`CANIF_E_FULL_TX_BUFFER`	`Assigned by DEM`

7.27 Error detection

CANIF018: The detection of development errors is configurable (*ON / OFF*) at pre-compile time. The switch `CANIF_DEV_ERROR_DETECT` (see chapter [10 Configuration specification]) shall activate or deactivate the detection of all development errors.

CANIF019: If the `CANIF_DEV_ERROR_DETECT` switch is enabled API checking is enabled. The detailed description of the detected errors can be found in chapter [7.26 Error classification] and chapter [8 API specification].

CANIF155: The detection of production code errors cannot be switched off.

CANIF295: When requested API operations are not supported, those errors are recognized as development error by `CANIF_E_NOK_NOSUPPORT`.

7.28 Error notification

CANIF156: Detected development errors shall be reported to `Det_ReportError` service of the Development Error Tracer (DET), if the pre-processor switch `CANIF_DEV_ERROR_DETECT` is set (see chapter 10).

CANIF020: Production errors shall be reported to the Diagnostic Event Manger (DEM). They shall not be used as the return value of the called function.

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

CANIF119: Additional errors that are detected because of specific implementation and/or specific hardware properties shall be added in the CAN Interface specific implementation specification. The classification and enumeration shall be compatible to the errors listed above.

7.29 Code version check

CANIF021: The CAN Interface files checks the consistency between the header, C and configuration files during compilation according to BSW004. This is to guarantee the consistency of the files and the code generator to the same release.

8 API specification

8.1 Imported types

CANIF142:

8.1.1 Standard types

In this chapter all used types included from the Std_Types.h are listed:

- Std_ReturnType
- Std_VersionInfoType

8.1.2 COM specific types

In this chapter all used types included from the ComStackTypes.h are listed:

- PduIdType
- PduLengthType
- PduInfoType

8.1.3 EcuM specific types

The following type of the EcuM shall be used:

- EcuM_WakeupSourceType

8.1.4 CAN specific types

The following type of the CAN Driver shall be used:

- Can_IdType
- Can_PduType

8.2 Type definitions

8.2.1 CanIf_ConfigType

CANIF144:

Type:	`struct`	
Range:	`Implementation Specific`	The contents of the initialization data structure are CAN Interface specific.
Description:	This type of the external data structure shall contain the post build initialization data for the CAN Interface for all underlying CAN Drivers.	

	The definition of CAN Interface public parameters shall contain: • Number of transmit L-PDUs • Number of receive L-PDUs • Number of dynamic transmit L-PDU handles The definition for each L-PDU handles shall contain: • Handle for transmit L-PDUs • Handle for receive L-PDUs • Name for transmit L-PDUs • Name for receive L-PDUs • CAN Identifier for static and dynamic transmit L-PDUs • CAN Identifier for receive L-PDUs • DLC for transmit L-PDUs • DLC for receive L-PDUs • Data buffer for receive L-PDUs in case of polling mode • Network towards each L-PDU belongs to. • Transmit L-PDU handle type

8.2.2 CanIf_ControllerConfigType

CANIF145:

Type:	`Struct`	
Range:	`Implementation Specific`	The contents of the initialization data structure are CAN Interface specific for initialization of all CAN controllers related to the CAN network.
Description:	This type of the external data structure shall contain the post build initialization data for the CAN Interface for all underlying CAN Drivers. The definition of CAN Interface public parameters shall contain: • Number of transmit L-PDUs • Number of receive L-PDUs • Number of dynamic transmit L-PDU handles The definition for each L-PDU handles shall contain: • Handle for transmit L-PDUs • Handle for receive L-PDUs • Name for transmit L-PDUs • Name for receive L-PDUs • CAN Identifier for transmit L-PDUs • CAN Identifier for receive L-PDUs • DLC for transmit L-PDUs • DLC for receive L-PDUs • Data buffer for receive L-PDUs in case of polling mode • Network towards each L-PDU belongs to. • Transmit L-PDU handle type Attention: dynamic transmit L-PDUs are not part of this type definition.	

8.2.3 CanIf_ControllerModeType

CANIF136:

Type:	Enumeration

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

8.2.4 CanIf_ChannelSetModeType

CANIF137:

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

8.2.5 CanIf_ChannelGetModeType

CANIF138:

Type:	Enumeration	
Range:	`CANIF_GET_OFFLINE = 0`	Channel is in the offline mode => no transmission and reception
	`CANIF_GET_RX_ONLINE`	Receive path of the corresponding channel is enabled and transmit path is disabled.
	`CANIF_GET_TX_ONLINE`	Transmit path of the corresponding channel is enabled and receive path is disabled.
	`CANIF_GET_ONLINE`	Channel is in the online mode => full operation mode

	CANIF_GET_OFFLINE_ACTIVE	Transmit path of the corresponding channel is in the offline active mode => transmit notifications are processed but transmit requests are blocked. The receive path is disabled.
	CANIF_GET_OFFLINE_ACTIVE_RX_ONLINE	Transmit path of the corresponding channel is in the offline active mode => transmit notifications are processed but transmit requests are blocked. The receive path is enabled.
Description:	Status of the PDU channel group. Current mode of the channel defines its transmit or receive activity. Communication direction (transmission and/or reception) of the channel can be controlled separately or together by upper layers.	

8.2.6 CanIf_NotifStatusType

CANIF201:

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

8.2.7 CanIf_TransceiverModeType

CANIF263:

Type:	Enumeration	
	CANIF_TRCV_MODE_NORMAL = 0	Transceiver mode NORMAL
	CANIF_TRCV_MODE_STANDBY	Transceiver mode STANDBY
	CANIF_TRCV_MODE_SLEEP	Transceiver mode SLEEP
Description:	Operating modes of the CAN Transceiver Driver.	

8.2.8 CanIf_TrcvWakeupReasonType

CANIF264:

Type:	Enumeration	
Range:	CANIF_TRCV_WU_ERROR = 0	Due to an error wake up reason was not detected. This value may only be reported when error was reported to DEM before.
	CANIF_TRCV_WU_NOT_SUPPORTED	The transceiver does not support any information for the wake up reason.
	CANIF_TRCV_WU_BY_BUS	The transceiver has detected, that the network has caused the wake up of the ECU.
	CANIF_TRCV_WU_INTERNALLY	The transceiver has detected, that the network has woken up by the ECU via a request to NORMAL mode.
	CANIF_TRCV_WU_RESET	The transceiver has detected, that the “wake up” is due to an ECU reset.

	CANIF_TRCV_WU_POWER_ON	The transceiver has detected, that the "wake up" is due to an ECU reset after power on.
Description:	This type shall be used to specify the wake up reason detected by the CAN transceiver in detail.	

8.2.9 CanIf_TrcvWakeupModeType

CANIF275:

Type:	Enumeration	
Range:	CANIF_TRCV_WU_ENABLE = 0	The notification for wakeup events is enabled on the addressed network.
	CANIF_TRCV_WU_DISABLE	The notification for wakeup events is disabled on the addressed network.
	CANIF_TRCV_WU_CLEAR	A stored wakeup event is cleared on the addressed network.
Description:	This type shall be used to specify the wake up reason detected by the CAN transceiver in detail.	

8.3 Function definitions

8.3.1 CanIf_Init

Service name:	CanIf_Init

Syntax:	`void CanIf_Init`
	`(`
	`  const CanIf_ConfigType  *ConfigPtr`
	`)`

Service ID:	0x01
Sync/Async:	Synchronous
Reentrancy:	Non re-entrant
	`ConfigPtr` Pointer to configuration parameter set, used e.g. for post build parameters
Parameters (out):	--
Return value:	--
Description:	**CANIF001:** This service initializes internal and external interfaces of the CAN Interface for the further processing. All underlying CAN controllers and CAN transceivers still remain not operational. This service is called only ECU State Manager (EcuM). If a NULL pointer is passed for `*ConfigPtr` to this function the default configuration shall be used. In case only one configuration setup is used, a NULL pointer is sufficient to choose the one static existing configuration setup. Development errors: - Invalid values of `*ConfigPtr` will be reported to the development error tracer (`CANIF_E_PARAM_POINTER`) only for post built use cases.
Caveats:	--
Configuration:	--

8.3.2 CanIf_InitController

Service name:	CanIf_InitController	
Syntax:	`void CanIf_InitController ( uint8 Controller, uint8 ConfigurationIndex )`	
Service ID:	0x02	
Sync/Async:	Synchronous	
Reentrancy:	Non re-entrant	
Parameters (in):	`Controller`	CAN controller requested for initialization
	`ConfigurationIndex`	Index to controller related configuration setup
Parameters (out):	--	
Return value:	--	
Description:	**CANIF002:** This service initializes in the CAN Interface the configured buffers of all Tx/Rx L-PDUs of the corresponding CAN controller. Different sets of static configuration may have been configured. A logical number is assigned to each set statically. The parameter `ConfigurationIndex` selects the configuration set that is used for initialization. The CAN controller still remains not operational and neither sends nor receives CAN L-PDUs. **CANIF022**: Development errors: - Invalid values of `Controller` or `ConfigurationIndex` will be reported to the development error tracer (`CANIF_E_PARAM_CONTROLLER` or `CANIF_E_PARAM_POINTER`) only for post built use cases. - If the CAN Interface was not initialized before invoking of `CanIf_Init()`, the call of this function will be reported to the development error tracer (`CANIF_E_UNINIT`). No initialization will be executed.	
Caveats:	CAN identifiers of dynamic transmit L-PDUs are not initialized by invoking this API.	
Configuration:	ID of the CAN controller is published inside the configuration description of the CAN Interface. At configuration time the relation has to be set up between the CAN Interface configuration set and the available corresponding CAN controller configuration sets by the CAN Driver configuration.	

8.3.3 CanIf_SetControllerMode

Service name:	CanIf_SetControllerMode	
Syntax:	`Std_ReturnType CanIf_SetControllerMode ( uint8 Controller, CanIf_ControllerModeType ControllerMode )`	
Service ID:	0x03	
Sync/Async:	Asynchronous	
Reentrancy:	Non re-entrant	
Parameters (in):	`Controller`	CAN controller requested for mode transition
	`ControllerMode`	Requested mode transition
Parameters (out):	--	
Return value:	`E_OK`	Network mode request has been accepted
	`E_NOT_OK`	Network mode request has not been accepted

Description:	**CANIF003:** This service calls the corresponding CAN Driver service for changing of the CAN controller mode. It initiates a transition to the requested CAN controller mode of one or multiple CAN controllers. This service calls `Can_SetControllerMode(Controller, Transition)` for the requested CAN controller. Development errors: - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (`CANIF_E_UNINIT`). The function returns with `E_NOT_OK`. - Invalid values of `Controller` will be reported to the development error tracer (`CANIF_E_PARAM_CONTROLLER`) only for post built use cases.
Caveats:	Re-entrant calls of this API are allowed only for different controller Identifiers. The CAN Driver must be initialized after Power ON. The CAN Interface must be initialized after Power ON.
Configuration:	ID of the CAN controller is published inside the configuration description of the CAN Interface.

8.3.4 CanIf_GetControllerMode

Service name:	CanIf_GetControllerMode	
Syntax:	`Std_ReturnType CanIf_GetControllerMode ( uint8 Controller, CanIf_ControllerModeType *ControllerModePtr )`	
Service ID:	0x04	
Sync/Async:	Synchronous	
Reentrancy:	Non re-entrant	
Parameters (in):	`Controller`	CAN controller requested for current operation mode
Parameters (out):	`CanIf_ControllerModePtr`	Pointer to a memory location, where the current mode of the CAN network will be stored.
Return value:	`E_OK`	Controller mode request has been accepted
	`E_NOT_OK`	Controller mode request has not been accepted
Description:	**CANIF229:** Service reports about the current status of the requested CAN controller. Development errors: - Invalid values of `Controller` will be reported to the development error tracer (`CANIF_E_PARAM_CONTROLLER`). - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (`CANIF_E_UNINIT`). The function returns with `E_NOT_OK`.	
Caveats:	The CAN Driver must be initialized after Power ON. The CAN Interface must be initialized after Power ON.	
Configuration:	ID of the CAN controller is published inside the configuration description of the CAN Interface Layer.	

8.3.5 CanIf_Transmit

Service name:	CanIf_Transmit

Syntax:	`Std_ReturnType CanIf_Transmit ( PduIdType CanTxPduId, const PduInfoType *PduInfoPtr )`	
Service ID:	0x05	
Sync/Async:	Synchronous	
Reentrancy:	Re-entrant	
Parameters (in)	`CanTxPduId`	L-PDU handle of CAN L-PDU to be transmitted. This handle specifies the corresponding CAN L-PDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device.
	`PduInfoPtr`	Pointer to a structure with CAN L-PDU related data: DLC and pointer to CAN L-SDU buffer
Parameters (out):	--	
Return value:	`E_OK`	Transmit request has been accepted
	`E_NOT_OK`	Transmit request has not been accepted
Description:	**CANIF005:** This service initiates a request for transmission of the CAN L-PDU specified by the `CanTxPduId` and CAN related data in the L-PDU structure. The corresponding CAN controller and HTH have to be resolved by the `CanTxPduId`. A transmit request has not been accepted, if the controller mode is not STARTED and/or the channel mode at least for the transmit path is not online or offline active. One call of this function results in one call of `Can_Write(Hth, *PduInfo)`. Development errors: - Invalid values of `CanTxPduId` or `PduInfoPtr` will be reported to the development error tracer (`CANIF_E_INVALID_TXPDUID` or `CANIF_E_PARAM_POINTER`). - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (`CANIF_E_UNINIT`). The function returns with `E_NOT_OK`.	
Caveats:	During the call of this API the buffer of PduInfoPtr is controlled by the CAN Interface may not be accessed for read/write from another call context. After return of this call the ownership changes to the upper layer. The CAN Interface must be initialized after Power ON.	
Configuration:	--	

8.3.6 CanIf_ReadRxPduData

Service name:	CanIf_ReadRxPduData	
Syntax:	`Std_ReturnType CanIf_ReadRxPduData ( PduIdType CanRxPduId, PduInfoType *PduInfoPtr )`	
Service ID:	0x06	
Sync/Async:	Synchronous	
Reentrancy:	Non re-entrant	
Parameters (in)	`CanRxPduId`	Receive L-PDU handle of CAN L-PDU. This handle specifies the corresponding CAN L-PDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device.
Parameters (out):	`PduInfoPtr`	Pointer to a structure with CAN L-PDU related data: DLC

		and pointer to CAN L-SDU buffer
Return value:	E_OK	Request for L-PDU data has been accepted
	E_NOT_OK	No valid data has been received.
Description:	**CANIF194:** This service provides the CAN DLC and the received data of the requested CanRxPduId to the calling upper layer. A request has not been accepted, if the network mode is not STARTED and/or the channel mode at least for the receive path online or offline active. Development errors: - Invalid values of CanRxPduId or PduInfoPtr will be reported to the development error tracer (CANIF_E_INVALID_RXPDUID or CANIF_E_PARAM_POINTER). - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (CANIF_E_UNINIT). The function returns with E_NOT_OK.	
Caveats:	During the call of this API the buffer of PduInfoPtr is controlled by the CAN Interface may not be accessed for read/write from another call context. After return of this call the ownership changes to the upper layer. This API must not be used for CanRxPduIds, which are defined to receive multiple CAN-Ids (range reception). The CAN Interface must be initialized after Power ON.	
Configuration:	This API can be enabled or disabled at pre-compile time configuration by the configuration parameter CANIF_READRXPDU_DATA_API.	

8.3.7 CanIf_ReadTxNotifStatus

Service name:	CanIf_ ReadTxNotifStatus	
Syntax:	CanIf_NotifStatusType CanIf_ReadTxNotifStatus (PduIdType CanTxPduId)	
Service ID:	0x07	
Sync/Async:	Synchronous	
Reentrancy:	Non re-entrant	
Parameters (in)	CanTxPduId	L-PDU handle of CAN L-PDU to be transmitted. This handle specifies the corresponding CAN L-PDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device.
Parameters (out):	--	
Return value:	CanIf_NotifStatusType	Current notification status of the corresponding CAN L-PDU.
Description:	**CANIF202:** This service provides the status of the static or dynamic CAN Tx L-PDU requested by CanTxPduId. This API service notifies the upper layer about any transmit confirmation event to the corresponding requested CAN L-PDU. During this call the CAN Tx L-PDU notification status is reset inside the CAN Interface. Development errors: - Invalid values of CanTxPduId will be reported to the development error tracer (CANIF_E_INVALID_TXPDUID). Error cases: ⇨ CanTxPduId is out of range or ⇨ no status information was configured for this CAN Tx L-PDU. - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (CANIF_E_UNINIT). The function returns with E_NOT_OK.	

Caveats:	The CAN Interface must be initialized after Power ON.
Configuration:	This API can be enabled or disabled at pre-compile time configuration globally by the parameter `CANIF_READTXPDU_NOTIFY_STATUS_API`.

8.3.8 CanIf_ReadRxNotifStatus

Service name:	CanIf_ ReadRxNotifStatus	
Syntax:	`CanIf_NotifStatusType CanIf_ReadRxNotifStatus ( PduIdType CanRxPduId )`	
Service ID:	0x08	
Sync/Async:	Synchronous	
Reentrancy:	Non re-entrant	
Parameters (in)	`CanRxPduId`	L-PDU handle of CAN L-PDU to be received. This handle specifies the corresponding CAN L-PDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device.
Parameters (out):	--	
Return value:	`CanIf_NotifStatusType`	Current notification status of the corresponding CAN Rx L-PDU.
Description:	**CANIF230:** This service provides the status of the CAN Rx L-PDU requested by CanRxPduId. This API service notifies the upper layer about any receive indication event to the corresponding requested CAN L-PDU. During this call the CAN Rx L-PDU notification status is reset inside the CAN Interface. Development errors: - Invalid values of `CanRxPduId` will be reported to the development error tracer (`CANIF_E_INVALID_RXPDUID`). Error cases: ⇨ `CanRxPduId` is out of range or ⇨ Status for `CanRxPduId` was requested whereas `CANIF_READRXPDU_DATA_API` is disabled ⇨ no status information was configured for this CAN Rx L-PDU. - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (`CANIF_E_UNINIT`). The function returns with `E_NOT_OK`.	
Caveats:	This API must not be used for CanRxPduIds, which are defined to receive multiple CAN-Ids (range reception). The CAN Interface must be initialized after Power ON.	
Configuration:	This API can be enabled or disabled at pre-compile time configuration globally by the parameter `CANIF_READRXPDU_NOTIFY_STATUS_API`.	

8.3.9 CanIf_SetPduMode

Service name:	CanIf_SetPduMode
Syntax:	`Std_ReturnType CanIf_SetPduMode ( uint8 Controller, CanIf_ChannelSetModeType PduModeRequest )`
Service ID:	0x09
Sync/Async:	Synchronous

Reentrancy:	Non re-entrant	
Parameters (in)	`Controller`	All PDUs of the own ECU connected to the corresponding physical CAN controller are addressed.
	`PduModeRequest`	Requested PDU mode change (see CanIf_ChannelSetModeType)
Parameters (out):	--	
Return value:	`E_OK`	Request for mode transition has been accepted
	`E_NOT_OK`	Request for mode transition has not been accepted
Description:	**CANIF008:** This service sets the requested mode at all L-PDUs of the predefined logical PDU channel. This channel parameter can be derived from Controller. Development errors: - Invalid values of `Controller` will be reported to the development error tracer (`CANIF_E_PARAM_CONTROLLLER`). - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (`CANIF_E_UNINIT`). The function returns with `E_NOT_OK`.	
Caveats:	Re-entrant calls of this API are allowed only for different channel Identifiers. The CAN Interface must be initialized after Power ON.	
Configuration:	The channel mode is configurable by `CANIF_CANTXPDUID_CONTROLLER/ CANIF_CANRXPDUID_CONTROLLER`.	

8.3.10 CanIf_GetPduMode

Service name:	CanIf_GetPduMode	
Syntax:	`Std_ReturnType CanIf_GetPduMode ( uint8 Controller, CanIf_PduGetModeType *PduModePtr )`	
Service ID:	0x0A	
Sync/Async:	Synchronous	
Reentrancy:	Non re-entrant	
Parameters (in)	`Controller`	All PDUs of the own ECU connected to the corresponding physical CAN controller are addressed.
Parameters (out):	`PduModePtr`	Pointer to a memory location, where the current mode of the logical PDU channel will be stored.
Return value:	`E_OK`	Pdu mode request has been accepted
	`E_NOT_OK`	Pdu request has not been accepted
Description:	**CANIF009:** This service reports the current mode of the requested Pdu channel Development errors: - Invalid values of `Controller` will be reported to the development error tracer (`CANIF_E_PARAM_CONTROLLER`). - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (`CANIF_E_UNINIT`). The function returns with `E_NOT_OK`.	
Caveats:	The CAN Interface must be initialized after Power ON.	
Configuration:	--	

8.3.11 CanIf_GetVersionInfo

Service name:	CanIf_GetVersionInfo
Syntax:	`void CanIf_GetVersionInfo ( Std_VersionInfoType *VersionInfo )`
Service ID [hex]:	0x0B
Sync/Async:	Synchronous
Reentrancy:	Non re-entrant
Parameters (in):	--
Parameters (out):	`VersionInfo` Pointer to where to store the version information of this module.
Return value:	--
Description:	**CANIF158:** This service returns the version information of this module. The version information includes: - Module Id - Vendor Id - Vendor specific version numbers (BSW00407). If source code for caller and callee of this function is available this function should be realized as a macro. The macro should be defined in the modules header file.
Caveats:	--
Configuration:	This function shall be pre compile time configurable `On/Off` by the configuration parameter `CANIF_VERSION_INFO_API`.

8.3.12 CanIf_SetDynamicTxId

Service name:	CanIf_SetDynamicTxId
Syntax:	`void CanIf_SetDynamicTxId ( PduIdType CanTxPduId, Can_IdType CanId )`
Service ID [hex]:	0x0C
Sync/Async:	Synchronous
Reentrancy:	Non re-entrant
Parameters (in):	`CanTxPduId` L-PDU handle of CAN L-PDU for transmission. This ID specifies the corresponding CAN L-PDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device. `CanId` Standard/Extended CAN ID of CAN L-PDU that shall be transmitted
Parameters (out):	--
Return value:	--
Description:	**CANIF189:** This service reconfigures the corresponding CAN identifier of the requested CAN L-PDU. Development errors: - Invalid values of CanTxPduId and CanId will be reported to the development error tracer (`CANIF_E_INVALID_TXPDUID` or `CANIF_E_PARAM_CANID`) - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (`CANIF_E_UNINIT`). No

	reconfiguration of Tx CanId will be executed.
Caveats:	The CAN Interface must be initialized after Power ON. This function may not be interrupted by `CanIf_Transmit()`, if the same L-PDU ID is handled.
Configuration:	This function shall be pre compile time configurable `On/Off` by the configuration parameter `CANIF_SETDYNAMICTXID_API`.

8.3.13 CanIf_SetTransceiverMode

Service name:	CanIf_SetTransceiverMode	
Syntax:	`Std_ReturnType CanIf_SetTransceiverMode ( uint8 Transceiver, CanIf_TransceiverModeType TransceiverMode )`	
Service ID [hex]:	0x0D	
Behavior:	Synchronous	
Reentrancy:	Non re-entrant	
Parameters (in):	`Transceiver`	CAN transceivers requested for mode transition
	`TransceiverMode`	Requested mode transition
Parameters (out):	--	
Return value:	`E_OK`	Will be returned, if the transceiver state has been changed to the requested mode.
	`E_NOT_OK`	Will be returned, if the transceiver state change has failed or the parameter is out of the allowed range. The previous state has not been changed.
Description:	**CANIF287:** This API requests actual state of CAN Transceiver Driver. For more details, please refer to the [9] Specification of CAN Transceiver Driver. This service calls `CanTrcv_SetOpMode (Transceiver, *OpMode)` for the corresponding requested CAN transceiver. Development errors: - Invalid values of `transceiver` or `transceiver mode` will be reported to the development error tracer (`CANIF_TRCV_E_TRANSCEIVER`, `CANIF_TRCV_E_TRCV_NOT_STANDBY` or `CANIF_TRCV_E_TRCV_NOT_NORMAL`) - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (`CANIF_E_UNINIT`). The function returns with `E_NOT_OK`.	
Caveats:	This API shall be applicable to all CAN transceivers with all values independent, if the transceiver hardware supports these modes or not. This is to ease up the view of the Can Interface to the assigned physical CAN channel. If the mode is not supported, the return value shall be E_OK.	
Configuration:	The number of supported transceiver types for each network is set up in the configuration phase. If no transceiver is used, this API shall not be provided.	

8.3.14 CanIf_GetTransceiverMode

Service name:	CanIf_GetTransceiverMode
Syntax:	`Std_ReturnType CanIf_GetTransceiverMode ( uint8 Transceiver,`

	`CanIf_TransceiverModeType *TransceiverModePtr` `)`	
Service ID [hex]:	0x0E	
Behavior:	Synchronous	
Reentrancy:	Non re-entrant	
Parameters (in):	`Transceiver`	CAN transceivers requested for mode transition
Parameters (out):	`TransceiverModePtr`	Requested mode transition
Return value:	`E_OK`	Transceiver mode request has been accepted
	`E_NOT_OK`	Transceiver mode request has not been accepted
Description:	**CANIF288:** This API returns actual state of CAN Transceiver Driver. For more details, please refer to the [9] Specification of CAN Transceiver Driver. This service calls `CanTrcv_GetOpMode (Transceiver, *OpMode)` for the corresponding requested CAN transceiver. Development errors: - Invalid values of `transceiver` will be reported to the development error tracer (`CANIF_TRCV_E_TRANSCEIVER`) - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (`CANIF_E_UNINIT`). The function returns with `E_NOT_OK`.	
Caveats:	See CanIf_Init() for the provided state after the CAN Transceiver Driver initialization till the first operation mode change request.	
Configuration:	The number of supported transceiver types for each network is set up in the configuration phase. If no transceiver is used, this API shall not be provided.	

8.3.15 CanIf_GetTrcvWakeupReason

Service name:	CanIf_GetTrcvWakeupReason	

Syntax:	`Std_ReturnType CanIf_GetTrcvWakeupReason`
	`(`
	`uint8 Transceiver`
	`CanIf_TrcvWakeupReasonType *TrcvWuReasonPtr`
	`)`	

Service ID [hex]:	0x0F	
Behavior:	Synchronous	
Reentrancy:	Non re-entrant	
Parameters (in):	`Transceiver`	The handle identifies the CAN transceiver to which the API call has to be applied.
Parameters (out):	`TrcvWuReasonPtr`	Requested transceiver wakeup reason
Return value:	`E_OK`	Transceiver mode request has been accepted
	`E_NOT_OK`	Transceiver mode request has not been accepted
Description:	**CANIF289:** This API returns the reason for the wake up that the CAN transceiver has detected. The ability to detect and differentiate the possible wakeup reasons depends strongly on the CAN transceiver hardware. For more details, please refer to the [9] Specification of CAN Transceiver Driver. This service calls `CanTrcv_GetBusWuReason (Transceiver, Reason)` for the corresponding requested CAN transceiver. Development errors:	

	- CANIF_TRCV_E_TRCV_NOT_STANDBY - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (CANIF_E_UNINIT). The function returns with E_NOT_OK.
Caveats:	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 has CAN, a wake up by CAN may occur and the incoming data may cause an internal wake up for another CAN network. This API has a "per network" view and does not vote the more important reason or sequence internally. The same may be true if e.g. one transceiver controls the power supply and the other is just powered or un-powered. Then one may be able to return CANIF_TRCV_WU_POWER_ON, whereas the other may state e.g. CANIF_TRCV_WU_RESET. It is up to the EcuM to decide, how to handle that wake up information.
Configuration:	The number of supported transceiver types for each network is set up in the configuration phase. If no transceiver is used, this API shall not be provided.

8.3.16 CanIf_SetTransceiverWakeupMode

Service name:	CanIf_SetTransceiverWakeupMode	
Syntax:	`Std_ReturnType CanIf_SetTransceiverWakeupMode ( uint8 Transceiver CanIf_TrcvWakeupModeType TrcvWakeupMode )`	
Service ID [hex]:	0x10	
Behavior:	Synchronous	
Reentrancy:	Non re-entrant	
Parameters (in):	Transceiver	The handle identifies the CAN transceiver to which the API call has to be applied.
	TrcvWakeupMode	Requested transceiver wakeup reason
Parameters (out):	--	
Return value:	E_OK	Will be returned, if the wakeup state has been changed to the requested mode.
	E_NOT_OK	Will be returned, if the wakeup state change has failed or the parameter is out of the allowed range. The previous state has not been changed.
Description:	**CANIF290:** This API enables, disables and clears the notification for wakeup events on the addressed network. For more details, please refer to the [9] Specification of CAN Transceiver Driver. This service calls CanTrcv_SetWakeupMode (Transceiver, TrcvWakeupMode) for the corresponding requested CAN transceiver. Enabled: if the CAN Transceiver Driver has a stored wakeup event pending for the addressed network, the notification is executed within the API call or immediately after (depending on the implementation). Disabled: Ii it is required by the transceiver device and the underlying communication, the driver has to detect the wakeup events nevertheless and stores it internally to raise the event, when the wakeup notification is enabled again. Clear: Clearing of wakeup events have to be used, when the wake up notification is disabled to clear all stored wake up events under control of the higher layer.	

	Development errors: CANTRCV_E_UNINIT: not yet initialized
Caveats:	The implementation may be e.g. disabling the interrupt source for the wake up. If the interrupt is level triggered a pending interrupt is automatically stored and raised after enabling the notification again. It is very important not to lose wake up events during the disabled period.
Configuration:	The number of supported transceiver types for each network is set up in the configuration phase. If no transceiover is used, this API shall not be provided.

8.3.17 CanIf_CheckWakeup

Service name:	CanIf_CheckWakeup	
Syntax:	`Std_ReturnType CanIf_CheckWakeup ( EcuM_WakeupSourceType WakeupSource )`	
Service ID:	0x11	
Sync/Async:	Synchronous	
Reentrancy:	Re-entrant	
	`WakeupSource`	Source device, who initiated the wakeup event: CAN controller or CAN transceiver
Parameters (out):	--	
Return value:	`E_OK`	Will be returned, if the check wakeup request has been accepted.
	`E_NOT_OK`	Will be returned, if the check wakeup request has not been accepted.
Description:	**CANIF219:** This Service checks, whether an underlying CAN driver or CAN Transceiver driver already signals an wakeup event by the CAN network This service shall evaluate the `WakeupSource` parameter to get the information, which dedicate wakeup source needs to be checked, either a CAN transceiver or controller device. Depending on this information the function `CanIf_CheckWakeup` shall either call the function `Can_Cbk_CheckWakeup()` or `CanTrcv_CB_WakeupByBus()` with the parameter addressing the correct hardware device causing the wakeup event. If one of these called functions has detected a wakeup by CAN (return value E_OK), the service `CanIf_CheckWakeup()` shall call the API `EcuM_SetWakeupEvent()` for the respective Wakeup Source. This service is called by the ECU Firmware. In dependence of the parameter value the CAN Interface notifies the CAN Driver or the CAN Transceiver Driver about the wakeup event. This service is implemented by the CAN Interface. It is called in case of a mode change notification of the CAN controller or the CAN transceiver. Development errors: - Invalid values of `WakeupSource` will be reported to the development error tracer (`CANIF_E_PARAM_WAKEUPSOURCE`). - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (`CANIF_E_UNINIT`). The function returns with `E_NOT_OK`.	
Caveats:	The call context is either on interrupt level (interrupt mode) or on task level (polling mode).	

	The CAN Interface must be initialized after Power ON. This call-out service is re-entrant for multiple CAN controller usage.
Configuration:	This wake up service is configurable by `CANIF_WAKEUP_SUPPORT`, which depends on the used CAN controller type and the used wakeup strategy. This callback may not be supported, if no wakeup shall be used.

8.3.18 CanIf_CheckValidation

Service name:	`CanIf_CheckValidation`	
Syntax:	`Std_ReturnType CanIf_CheckValidation ( EcuM_WakeupSourceType WakeupSource )`	
Service ID:	0x12	
Sync/Async:	Synchronous	
Reentrancy:	Re-entrant	
Parameters (in):	`WakeupSource`	Source device, who initiated the wakeup event has to be validated: CAN controller or CAN transceiver
Parameters (out):	--	
Return value:	`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:	**CANIF178:** This service is performed to validate a previous wakeup event. This service is called by the ECU Firmware. The CAN Interface checks inside this service, whether a L-PDU was successful received inn the meantime. **CANIF179:** The validation call return, whether the first CAN L-PDU reception event after a wakeup event has been occurred on the corresponding CAN network. In that case `EcuM_ValidateWakeupEvent()` is called within the validation result. For different upper layer users different service names shall be used. This type of indication call-out service is mainly designed for the ECU State Manager module. Development errors: - Invalid values of `WakeupSource` will be reported to the development error tracer (`CANIF_E_PARAM_WAKEUPSOURCE`). - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (`CANIF_E_UNINIT`).). The function returns with `E_NOT_OK`.	
Caveats:	The CAN Driver must be initialized after Power ON. The call context is either on interrupt level (interrupt mode) or on task level (polling mode). This call-out service is re-entrant for multiple CAN controller/CAN network usage.	
Configuration:	The responsible layers for the copying of the data are statically configurable. If no validation is needed, this API can be omitted by disable of `CANIF_WAKEUP_VALIDATION`. The wakeup validation API name for validated wakeup events belonging to the EcuM module must be configured to `EcuM_ValidateWakeupEvent()`.	

8.4 Call-out notifications

This is a list of functions provided for other modules. The function prototypes of the callback functions shall be provided in the file `canif_cbk.h`.

8.4.1 CanIf_TxConfirmation

Service name:	CanIf_TxConfirmation
Syntax:	`void CanIf_TxConfirmation ( PduIdType CanTxPduId )`
Service ID:	0x13
Sync/Async:	Synchronous
Reentrancy:	Re-entrant
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 (out):	--
Return value:	--
Description:	**CANIF007:** This service is implemented in the CAN Interface and called by the CAN Driver after the CAN L-PDU has been transmitted on the CAN network. Within this service, the CAN Driver passes back the `CanTxPduId` to the CAN Interface, which it got from `Can_Write(Hth, *PduInfo).` This call-out service is implemented as many times as underlying CAN Drivers are used. In that case one transmit confirmation call-out is assigned to one underlying CAN Driver. Then following naming convention has to be considered: `CanIf_TxConfirmation_<CAN_Driver>`. For further details please refer to chapter [7.25 Multiple CAN Driver support]. Development errors: - Invalid values of `CanTxPduId` will be reported to the development error tracer (`CANIF_E_PARAM_LPDU`). - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (`CANIF_E_UNINIT`). No Tx confirmation handling will be executed.
Caveats:	The call context is either on interrupt level (interrupt mode) or on task level (polling mode). This call-out service is re-entrant for multiple CAN controller usage. The CAN Interface must be initialized after Power ON.
Configuration:	Transmit confirmation can be enabled or disabled by configuration. It is always enabled, if transmit buffers are used.

8.4.2 CanIf_RxIndication

Service name:	CanIf_RxIndication
Syntax:	`void CanIf_RxIndication ( uint8 Hrh, Can_IdType CanId, uint8 CanDlc, const uint8 *CanSduPtr`

	)	
Service ID:	0x14	
Sync/Async:	Synchronous	
Reentrancy:	Re-entrant	
Parameters (in):	`Hrh`	ID of the corresponding hardware object Range: • 0..(total number of Hardware Receive Handles – 1)
	`CanId`	Standard/Extended CAN ID of CAN L-PDU that has been successfully received
	`CanDlc`	Data length code (length of CAN L-PDU payload)
	`*CanSduPtr`	Pointer to received L-SDU (payload)
Parameters (out):	--	
Return value:	--	
Description:	**CANIF006:** This service is implemented in the CAN Interface and called by the CAN Driver after a CAN L-PDU has been received. Within this service, the CAN Interface translates the `CanId` into the configured target PDU ID and routes this indication to the configured upper layer target service(s). This call-out service is implemented as many times as underlying CAN Drivers are used. In that case one receive indication call-out is assigned to one underlying CAN Driver. Then following naming convention has to be considered: `CanIf_RxIndication_<CAN_Driver>`. For further details please refer to chapter [7.25 Multiple CAN Driver support]. Development errors: - Invalid values of `Hrh, CanId, CanDlc` or `*CanSduPtr` will be reported to the development error tracer (`CANIF_E_PARAM_HRH, CANIF_E_PARAM_CANID, CANIF_E_PARAM_DLC` or `CANIF_E_PARAM_POINTER`). - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (`CANIF_E_UNINIT`). No Rx indication handling will be executed.	
Caveats:	The call context is either on interrupt level (interrupt mode) or on task level (polling mode). This call-out service is re-entrant for multiple CAN controller usage. The CAN Interface must be initialized after Power ON.	
Configuration:	CAN L-PDUs have to be assigned to the corresponding receive indication service.	

8.4.3 CanIf_CancelTxConfirmation

Service name:	CanIf_CancelTxConfirmation	

Syntax:	`void CanIf_CancelTxConfirmation`
	`(`
	`  const Can_PduType  *PduInfoPtr`
	`)`	

Service ID:	0x15	
Sync/Async:	Synchronous	
Reentrancy:	Re-entrant	
Parameters (in)	`*PduInfoPtr`	Pointer to a structure with CAN L-PDU related data: L-PDU handle of the successfully aborted CAN L-PDU, CAN identifier, DLC and pointer to CAN L-SDU buffer.

Parameters (out):	--
Return value:	--
Description:	**CANIF101:** This service is implemented in the CAN Interface and called by the CAN Driver after a previous request for cancellation of a pending L-PDU transmit request was successfully performed. This callback service is implemented as many times as underlying CAN Drivers are used. In that case one cancel transmit confirmation callback is assigned to one underlying CAN Driver. Then following naming convention has to be considered: `CanIf_CancelTxConfirmation_<CAN_Driver>`. For further details please refer to chapter [7.25 Multiple CAN Driver support]. Development errors: - Invalid values of `CanTxPduId` will be reported to the development error tracer (`CANIF_E_PARAM_LPDU`). - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (`CANIF_E_UNINIT`). No Tx cancellation handling will be executed.
Caveats:	The call context is either on interrupt level (interrupt mode) or on task level (polling mode). The CAN Interface must be initialized after Power ON.
Configuration:	This function shall be pre compile time configurable `On/Off` by the configuration parameter `CANIF_TRANSMIT_CANCELLATION`.

8.4.4 CanIf_ControllerBusOff

Service name:	CanIf_ControllerBusOff

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

Service ID:	0x16
Sync/Async:	Synchronous
Reentrancy:	Re-entrant
Parameters (in):	`Controller` CAN controller, where a BusOff occurred.
Parameters (out):	--
Return value:	--
Description:	**CANIF218:** This service indicates a CAN controller BusOff event referring to the corresponding CAN controller. (CANIF014) This call-out service is called by the CAN Driver and implemented in the CAN Interface. It is called in case of a mode change notification of the CAN Driver. This call-out service is implemented as many times as underlying CAN Drivers are used. In that case one BusOff notification is assigned to one underlying CAN Driver. Then following naming convention has to be considered: `CanIf_ControllerBusOff_<CAN_Driver>`. For further details please refer to chapter [7.25 Multiple CAN Driver support]. Development errors: - Invalid values of `controller` will be reported to the development error tracer (`CANIF_E_PARAM_CONTROLLER`). - If the CAN Interface was not initialized before, the call of this function will be reported to the development error tracer (`CANIF_E_UNINIT`). No BusOff

	notification will be executed.
Caveats:	The call context is either on interrupt level (interrupt mode) or on task level (polling mode). The CAN Interface must be initialized after Power ON. This call-out service is re-entrant for multiple CAN controller usage.
Configuration:	ID of the CAN controller is published inside the configuration description of the CAN Interface.

8.5 Expected interfaces

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

8.5.1 Mandatory interfaces

CANIF040: This chapter defines all interfaces which are required to fulfill the core functionality of the module.

API function	***Module***	***Description***
`Can_InitController`	CAN Driver	Service for CAN controller specific initialization of the CAN Hardware unit.
`Can_SetControllerMode`	CAN Driver	Service to initiate state transitions of the corresponding CAN controller.
`Can_DisableControllerInterrupts`	CAN Driver	Service for disabling the interrupts of the CAN corresponding controller.
`Can_EnableControllerInterrupts`	CAN Driver	Service for enabling the interrupts of the CAN corresponding controller.
`Can_Cbk_CheckWakeup`	CAN Driver	Service to evaluate CAN controller device, which caused a wakeup
`Can_Write`	CAN Driver	Service for transmitting CAN L-PDUs.
`CanTrcv_CB_WakeupByBus`	Can Transceiver Driver	Service to evaluate CAN transceiver device, which caused a wakeup
`CanSM_ControllerBusOff`	CAN Station Manager	Service to notify CanSM about an BusOff event
`Dem_ReportErrorStatus`	DEM	Reporting of production errors. Function can also be used before DEM is initialized.

8.5.2 Optional interfaces

CANIF294: This chapter defines all interfaces which are required to fulfill an optional functionality of the module.

API function	***Module***	***Description***	***Configuration parameter (description see chapter 10)***
`CanTrcv_SetOpMode`	CanTrcv	Service to initiate state transitions of the corresponding CAN transceiver.	Configuration parameters in container CanInterfaceTransceiverDriverConfiguration

CanTrcv_GetOpMode	CanTrcv	Service to read the current state of the corresponding CAN transceiver.	Configuration parameters in container CanInterfaceTransceiverDriverConfiguration
CanTrcv_GetBusWuReason	CanTrcv	Service to read the last wakeup reason of the corresponding CAN transceiver.	Configuration parameters in container CanInterfaceTransceiverDriverConfiguration
CanTrcv_SetWakeupMode	CanTrcv	Service to initiate wakeup mode of the corresponding CAN transceiver.	Configuration parameters in container CanInterfaceTransceiverDriverConfiguration
CanTrcv_CheckWakeup	CanTrcv	Service to evaluate CAN transceiver device, which caused a wakeup	Configuration parameters in container CanInterfaceTransceiverDriverConfiguration
Det_ReportError	Det	Development error notification	CANIF_DEV_ERROR_DETECT

8.5.3 Configurable interfaces

In this chapter all interfaces are listed, where the target function of any upper layer to be called has to be set up by configuration. These call-out services are specified and implemented in the upper communication modules, which use the CAN Interface according to the AUTOSAR BSW architecture. The specific call-out 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 call-out is performed, if no API name is configured. This chapter describes only the content of notification of the call-out, the call context inside the CAN Interface 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 ('user') and called by the CAN Interface. This condition displays the symbolic name of the functional group in a call-out service in the corresponding upper layer. Each upper layer can define no, one or several call-out services for the same functionality (i.e. transmit confirmation). The dispatch is ensured by the L-PDU ID.

8.5.3.1 <User_TxConfirmation> (PDU Router, CanNm, CanTp)

Service name:	<User_TxConfirmation>

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

Service ID:	0x17
Sync/Async:	Synchronous
Reentrancy:	Re-entrant

Parameters (in)	`Can<User>PduId`	Target PDU ID of CAN L-PDU transmitted successfully. This handle specifies the corresponding CAN L-PDU ID and implicitly the CAN Driver instance as well as the corresponding CAN controller device. Range: 0..(maximum number of PDU IDs received) – 1
Parameters(out):	--	
Return value:	--	
Description:	**CANIF011:** This service confirms a previous successfully processed CAN transmit request. This call-out service is called by the CAN Interface and implemented by the corresponding upper layer. It is called in case of a transmit confirmation of the CAN Driver. This type of confirmation call-out service is mainly designed for the PDU Router, CanNm and CanTp module.	
Caveats:	The call context is either on interrupt level (interrupt mode) or on task level (polling mode). This call-out service is re-entrant for multiple CAN controller/CAN network usage.	
Configuration:	This call-out service has to be configured by `CANIF_USER_TX_CONFIRMATION`. If no upper layers are configured, no confirmation is executed. If `CANIF_TX_USER_TYPE` is set to PduR, CanNm or CanTp, the corresponding Tx confirmation call-out service defined and implemented in the corresponding upper layer module will be called. In this case `CANIF_USER_TX_CONFIRMATION` can be ignored.	

8.5.3.2 <User_RxIndication> (PDU Router)

Service name:	<User_RxIndication>	

Syntax:	`void <User_RxIndication>`
	`(`
	`  PduIdType  Can<User>RxPduId,`
	`  const uint8  *CanSduPtr`
	`)`	

Service ID:	0x18	
Sync/Async:	Synchronous	
Reentrancy:	Re-entrant	
Parameters (in)	`Can<User>RxPduId`	Target PDU ID of CAN L-PDU that has been received. This handle identifies the data that has been received. Range: 0..(maximum number of PDU IDs received by this upper layer) – 1
	`CanSduPtr`	Pointer to received SDU (payload buffer).
Parameters (out):	--	
Return value:	--	
Description:	**CANIF012:** This service indicates a successful reception of an L-PDU to e.g. the PDU Router after passing all filters and validation checks. This call-out service is called by the CAN Interface and implemented by the configured upper layer (PDU Router). It is called in case of a receive indication event (i.e. ISR is triggered) of the CAN Driver. The data shall be copied by the corresponding upper layer via `*CanSduPtr`. In this case the L-PDU buffers are not global and distributed in the corresponding upper layer. This type of indication call-out service is mainly designed for the PDU Router	

	module.
Caveats:	Until this service returns the CAN Interface will not access `*CanSduPtr`. The `*CanSduPtr` is only valid and can be used by upper layers until the indication returns. CAN Interface guarantees that the number of configured bytes for this `CanRxPduId` is valid. The CAN Driver must be initialized after Power ON. The call context is either on interrupt level (interrupt mode) or on task level (polling mode).This call-out service is re-entrant for multiple CAN controller/CAN network usage.
Configuration:	This call-out service has to be configured by `CANIF_USER_RX_INDICATION`. If no upper layers are configured, no indication is executed. If `CANIF_RX_USER_TYPE` is set to PduR, the corresponding Rx indication call-out service `PduR_CanIfRxIndication()`defined and implemented in the corresponding PduR module will be called. In this case `CANIF_USER_RX_INDICATION` can be ignored.

8.5.3.3 <User_RxIndication> (CanNm)

Service name:	<User_RxIndication>	

Syntax:	`void <User_RxIndication>`
	`(`
	`  PduIdType  Can<User>RxPduId,`
	`  const uint8  *CanSduPtr`
	`)`	

Service ID:	0x19	
Sync/Async:	Synchronous	
Reentrancy:	Re-entrant	
	`Can<User>RxPduId`	Target PDU ID of CAN L-PDU that has been received. This handle identifies the data that has been received. The value passed to CAN NM via the API parameter CanNmRxPduId shall refer to the logical network identifier. Range: 0..(maximum number of PDU IDs received) – 1
	`CanSduPtr`	Pointer to received L-SDU (payload).
Parameters (out):	--	
Return value:	--	
Description:	**CANIF221:** This service indicates a successful reception of a received L-PDU to the upper layer after passing all filters and validation checks. This call-out service is called by the CAN Interface and implemented by the configured upper layer (CanNm). It is called in case of a receive indication event (i.e. ISR is triggered) of the CAN Driver. The data is copied by the corresponding upper layer via `*CanSduPtr`. In this case the L-PDU buffers are not global and distributed in the corresponding upper layer. This type of indication call-out service is mainly designed for the CanNm module.	
Caveats:	Until this service returns the CAN Interface will not access `*CanSduPtr`. The `*CanSduPtr` is only valid and can be used by upper layers until the indication returns. CAN Interface guarantees that the number of configured bytes for this `Can<User>RxPduId` is valid. The CAN Driver must be initialized after Power ON.	

	The call context is either on interrupt level (interrupt mode) or on task level (polling mode). This call-out service is re-entrant for multiple CAN controller/CAN network usage.
Configuration:	This call-out service has to be configured by `CANIF_USER_RX_INDICATION`. This call-out service is mandatory If `CANIF_RX_USER_TYPE` is set to CanNm, the corresponding Rx indication call-out service `CanNm_RxIndication()`defined and implemented in the corresponding CanNm module will be called. In this case `CANIF_USER_RX_INDICATION` can be ignored.

8.5.3.4 <User_RxIndication> (CanTp)

Service name:	<User_RxIndication>	

Syntax:	`void <User_RxIndication>`
	`(`
	`  PduIdType  Can<User>RxPduId,`
	`  const PduInfoType  *PduInfoPtr`
	`)`	

Service ID [hex]:	0x1A	
Sync/Async:	Synchronous	
Reentrancy:	Re-entrant	
Parameters (in):	`Can<User>RxPduId`	Target PDU handle of CAN L-PDU that has been received. Identifies the data that has been received. Range: 0..(maximum number of PDU IDs received) – 1
	`PduInfoPtr`	Pointer to structure with received L-SDU (payload) and data length (DLC).
Parameters (out):	--	
Return value:	--	
Description:	**CANIF195:** This function is called by the CAN Interface after a successful reception of a receive CAN L-PDU belonging to e.g. the CanTp. This call-out service is called by the Can Interface and implemented the configured upper layer (CanTp). It shall be called in case of a receive indication of CAN Driver. The data shall be copied by the corresponding upper layer via the PDU structure `*PduInfoPtr`. In this case the L-PDU buffers are not global and distributed in the e.g. CAN Transport Layer. This type of indication call-out service is mainly designed for the CanTp module.	
Caveats:	Until this service returns the CAN Interface will not access `*PduInfoPtr`. The `*PduInfoPtr` is only valid and can be used by upper layers until the indication returns. CAN Interface guarantees that the number of configured bytes for this `Can<User>RxPduId` is valid. The CAN Driver must be initialized after Power ON. The call context is either on interrupt level (interrupt mode) or on task level (polling mode). This call-out service is re-entrant for multiple CAN controller/CAN network usage.	
Configuration:	This call-out service has to be configured by `CANIF_USER_RX_INDICATION`. If no upper layers are configured, no indication is executed. If `CANIF_RX_USER_TYPE` is set to CanTp, the corresponding Rx indication call-out service `CanTp_RxIndication()`defined and implemented in the corresponding CanTp module will be called. In this case `CANIF_USER_RX_INDICATION` can be ignored.	

8.5.3.5 <User_ControllerBusOff> (CanSM)

Service name:	<User_ControllerBusOff>

Syntax:	`void <User_ControllerBusOff>`
	`(`
	`  uint8  Controller`
	`)`

Service ID:	0x1B
Sync/Async:	Synchronous
Reentrancy:	Re-entrant
Parameters (in):	`Controller` CAN network, where at least at one CAN controller a BusOff occurred.
Parameters (out):	--
Return value:	--
Description:	**CANIF014:** This service indicates a BusOff event at the notified CAN controller device. This call-out service is called by the CAN Interface and implemented by the CAN State Manager as `CanSM_ControllerBusOff()` ([11] Specification of CAN State Manager) or any other upper layer. It is called in case of a BusOff notification `CanIf_ControllerBusOff()` of the CAN Driver. For different upper layer users different service names shall be used. This type of indication call-out service is mainly designed for the Communication Manager module.
Caveats:	The CAN Driver must be initialized after Power ON. The call context is either on interrupt level (interrupt mode) or on task level (polling mode). This call-out service is re-entrant for multiple CAN controller/CAN network usage. Before re-initialization/restart during BusOff recovery is executed this call-out service is performed only once in case of multiple BusOff events at CAN controllers of the corresponding CAN network.
Configuration:	ID of the CAN network is published inside the configuration description of the CAN Interface. This call-out service is mandatory and configured by `CANIF_USER BUSOFF_NOTIFICATION`. If the controller BusOff notification API name for BusOff events belonging to the CanSM module, it has to be configured to `CanSM_ControllerBusOff()`.

8.5.3.6 <User_SetWakeupEvent> (EcuM)

Service name:	<User_SetWakeupEvent>

Syntax:	`void <User_SetWakeupEvent>`
	`(`
	`  EcuM_WakeupSourceType  CanWakupEvents`
	`)`

Service ID:	0x1C
Sync/Async:	Synchronous
Reentrancy:	Re-entrant
Parameters (in):	`CanWakupEvents` Events to be validated. Every CAN network can be a separate wakeup source.
Parameters (out):	--
Return value:	--
Description:	**CANIF013:** Service indicates a wake up event initiated from the CAN network and detected by the CAN Driver ro CAN Transceiver Driver. This call-out service is called by the CAN Interface and implemented by the upper layer (`EcuM_SetWakeupEvent()` in ECU State Manager).

	It is called only during call of `CanIf_CheckWakeup()`. For different upper layer users different service names shall be used. This type of indication call-out service is mainly designed for the ECU State Manager module.
Caveats:	The CAN Driver must be initialized after Power ON. The call context is either on interrupt level (interrupt mode) or on task level (polling mode). This call-out service is re-entrant for multiple CAN controller/CAN network usage. Before re-initialization/restart is executed this call-out service is performed only once in case of multiple wakeup events at CAN controllers of the corresponding CAN network.
Configuration:	The responsible layers for the copying of the data are statically configurable. If no upper layer call-out is configured no notification is configured by `CANIF_WAKEUP_SUPPORT`. If the wakeup notification API name for wakeup events over CAN belonging to the EcuM module, it has to be configured to `EcuM_SetWakeupEvent()`.

8.5.3.7 <User_ValidationWakeupEvent> (EcuM)

Service name:	<User_ValidationWakeupEvent>

Syntax:	`void <User_ValidationWakeupEvent>`
	`(`
	`  EcuM_WakeupSourceType  CanWakeupEvents`
	`)`

Service ID:	0x1D
Sync/Async:	Synchronous
Reentrancy:	Re-entrant
Parameters (in):	`CanWakeupEvents` Validated CAN wakeup events. Every CAN network can be a separate wakeup source.
Parameters (out):	--
Return value:	--
Description:	**CANIF178:** This notification is performed, when a previous wakeup event has been validated. This call-out service is called by the CAN Interface and implemented by the upper layer (`EcuM_ValidateWakeupEvent()` in ECU State Manager). **CANIF179:** The validation call-out is performed, only during call of `CanIf_CheckValidation()` and whenever the first CAN L-PDU reception event after a wakeup event has been occurred on the corresponding CAN network. For different upper layer users different service names shall be used. This type of indication call-out service is mainly designed for the ECU State Manager module.
Caveats:	The CAN Driver must be initialized after Power ON. The call context is either on interrupt level (interrupt mode) or on task level (polling mode). This call-out service is re-entrant for multiple CAN controller/CAN network usage.
Configuration:	The responsible layers for the copying of the data are statically configurable. If no upper layer call-out is configured by `CANIF WAKEUP_VALIDATION_`, no notification is performed. If the wakeup validation API name for validated wakeup events belonging to the EcuM module, it has to be configured to `EcuM_ValidateWakeupEvent()`.

9 Sequence diagrams

The following sequence diagrams show the interaction between the CAN Interface and the CAN Driver.

9.1 Transmit request (single CAN Driver)

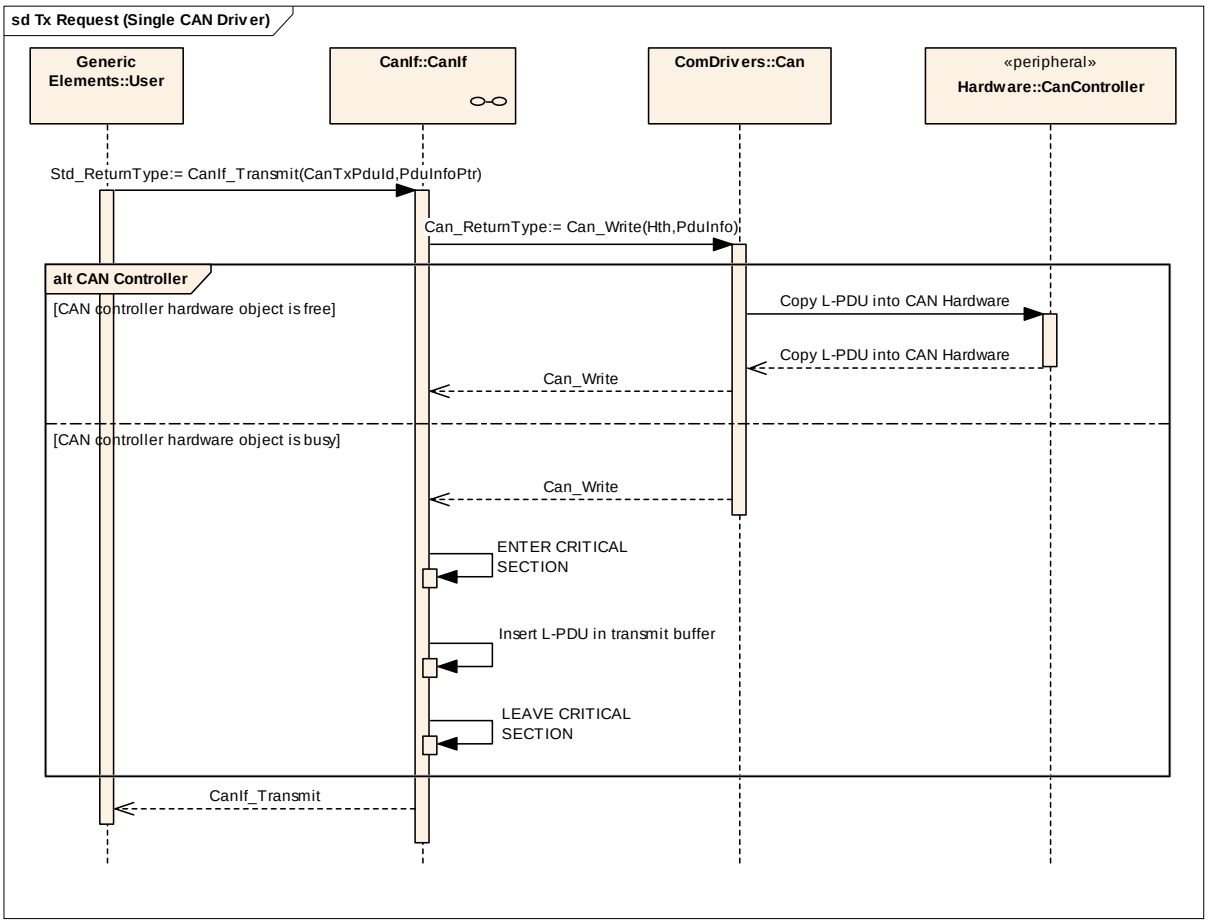

Figure 23 Transmission request with a single CAN Driver

Activity	*Description*
Transmission request	The upper layer initiates a transmit request via the service `CanIf_Transmit()`. The parameter CanTxPduId identifies the requested L-PDU. The service performs following steps: - validation of the input parameter - definition of the CAN controller to be used The second parameter *PduInfoPtr is a pointer on the structure with transmit L-PDU related data such as CanSduLength and *CanSduPtr.
Start transmission	`CanIf_Transmit()` requests a transmission and calls the CAN Driver service `Can_Write()` with corresponding processing of the HTH.
Hardware request	`Can_Write()`writes all L-PDU data in the CAN Hardware (if it is free) and sets the hardware request for transmission.
E_OK from Can_Write service	`Can_Write()` returns E_OK to `CanIf_Transmit()`.
E_BUSY from Can_Write service	If the CAN Driver detects, there are no free hardware objects available, it returns `CAN_E_BUSY` to the CAN Interface.
Copying into the buffer	The L-PDU of the rejected transmit request will be inserted in the transmit buffer of the CAN Interface until the next transmit confirmation.
E_OK from CAN Interface	`CanIf_Transmit()` returns E_OK to the upper layer.

9.2 Transmit request (multiple CAN Drivers)

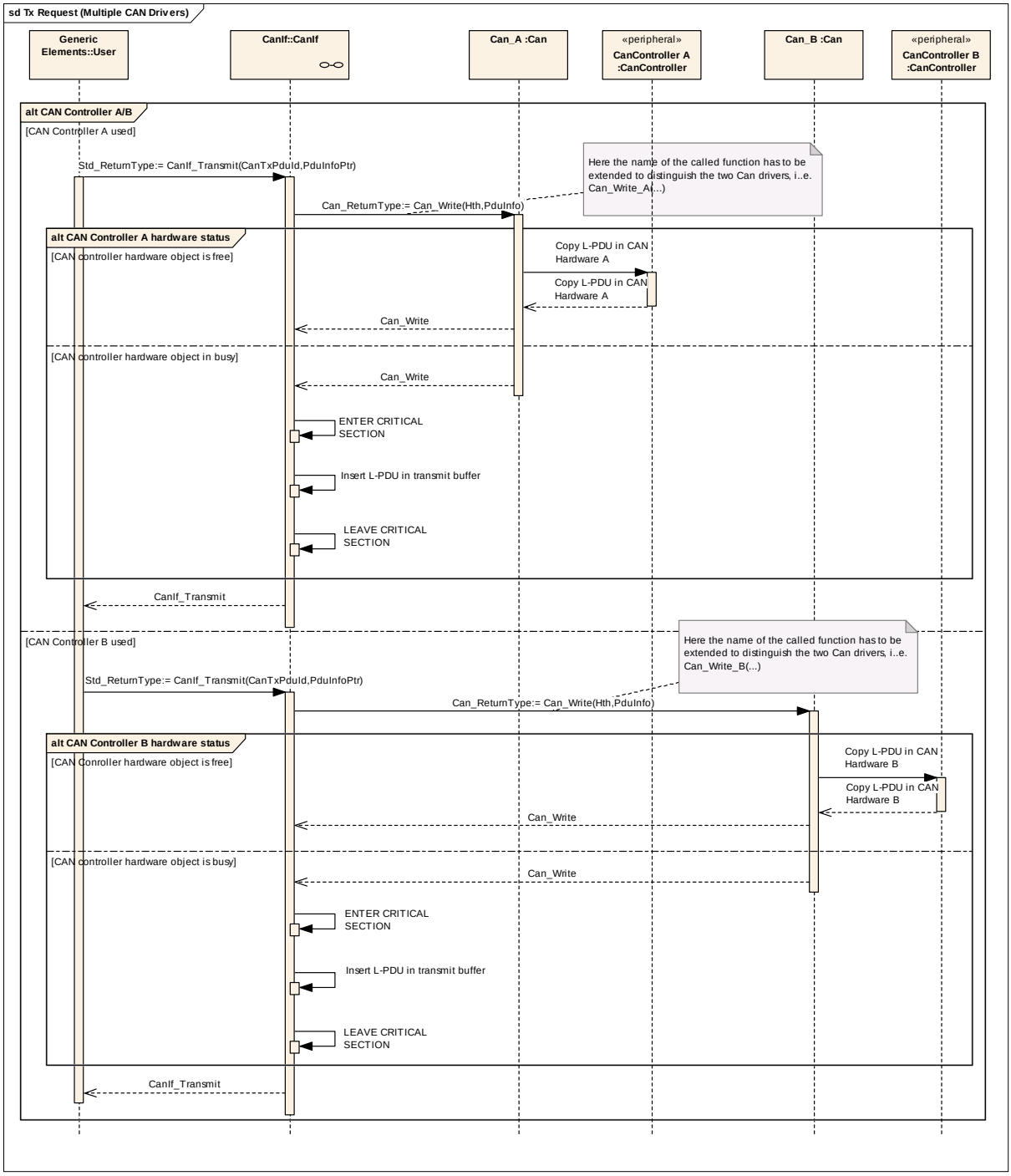

Figure 24 Transmission request with multiple CAN Drivers

First transmit request:

Activity	*Description*
Transmission request A	The upper layer initiates a transmit request via the service `CanIf_Transmit()`. The parameter CanTxPduId identifies the requested L-PDU. The service performs following steps: - validation of the input parameter - definition of the CAN controller to be used (here: A) The second parameter *PduInfoPtr is a pointer on the structure with transmit L-PDU related data such as CanSduLength and *CanSduPtr.
Start transmission	`CanIf_Transmit()` requests a transmission and calls the CAN Driver A service `Can_Write_A()` with corresponding processing of the HTH.
Hardware request	`Can_Write_A()`writes all L-PDU data in the CAN Hardware A (if it is free) and sets the hardware request for transmission.
E_OK from Can_Write service	`Can_Write_A()` returns E_OK to `CanIf_Transmit()`.
E_BUSY from Can_Write service	If the CAN Driver A detects, there are no free hardware objects available, it returns `CAN_E_BUSY` to the CAN Interface.
Copying into the buffer	The L-PDU of the rejected transmit request will be inserted in the transmit buffers of the CAN Interface until the next transmit confirmation.
E_OK from CAN Interface	`CanIf_Transmit()` returns E_OK to the upper layer.

Second transmit request:

Activity	*Description*
Transmission request B	The upper layer initiates a transmit request via the service `CanIf_Transmit()`. The parameter CanTxPduId identifies the requested L-PDU. The service performs following steps: - validation of the input parameter - definition of the CAN controller to be used (here: B) The second parameter *PduInfoPtr is a pointer on the structure with receive L-PDU related data such as CanSduLength and *CanSduPtr.
Start transmission	`CanIf_Transmit()` starts a transmission and calls the CAN Driver A service `Can_Write_B()` with corresponding processing of the HTH.
Hardware request	`Can_Write_B()`writes all L-PDU data in the CAN Hardware B (if it is free) and sets the hardware request for transmission.
E_OK from Can_Write service	`Can_Write_B()` returns E_OK to `CanIf_Transmit()`.
E_BUSY from Can_Write service	If the CAN Driver B detects, there are no free hardware objects available, it returns `CAN_E_BUSY` to the CAN Interface.
Copying into the buffer	The L-PDU of the rejected transmit request will be inserted in the transmit buffers of the CAN Interface until the next transmit confirmation.
E_OK from CAN Interface	`CanIf_Transmit()` returns E_OK to the upper layer.

9.3 Transmit confirmation (interrupt mode)

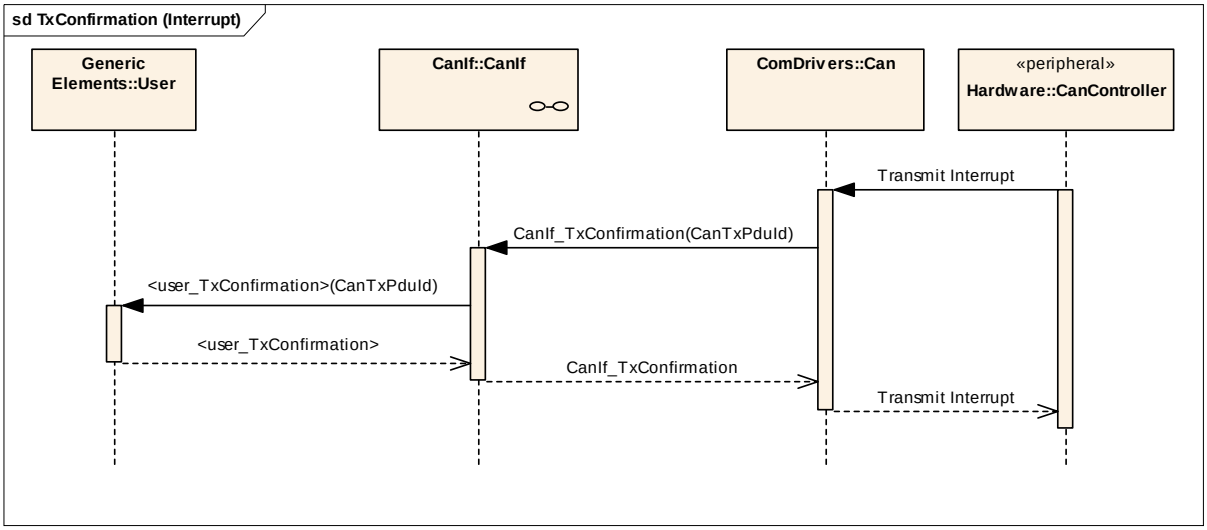

Figure 25 Transmit confirmation interrupt driven

Activity	***Description***
Transmit interrupt	The acknowledged CAN frame signals a successful transmission to the receiving CAN controller and triggers the transmit interrupt.
Confirmation to the CAN Interface	CAN Driver calls the service `CanIf_TxConfirmation()`. The parameter CanTxPduId specifies the CAN L-PDU previously sent by `Can_Write()`. The CAN diver must store the all in HTHs pending L-PDU Ids in an array organized per HTH to avoid new search of the L-PDU ID for call of `CanIf_TxConfirmation()`.
Confirmation to upper layer	Calling of the corresponding upper layer confirmation service `<User_TxConfirmation>()`. It signals a successful L-PDU transmission to the upper layer.

9.4 Transmit confirmation (polling mode)

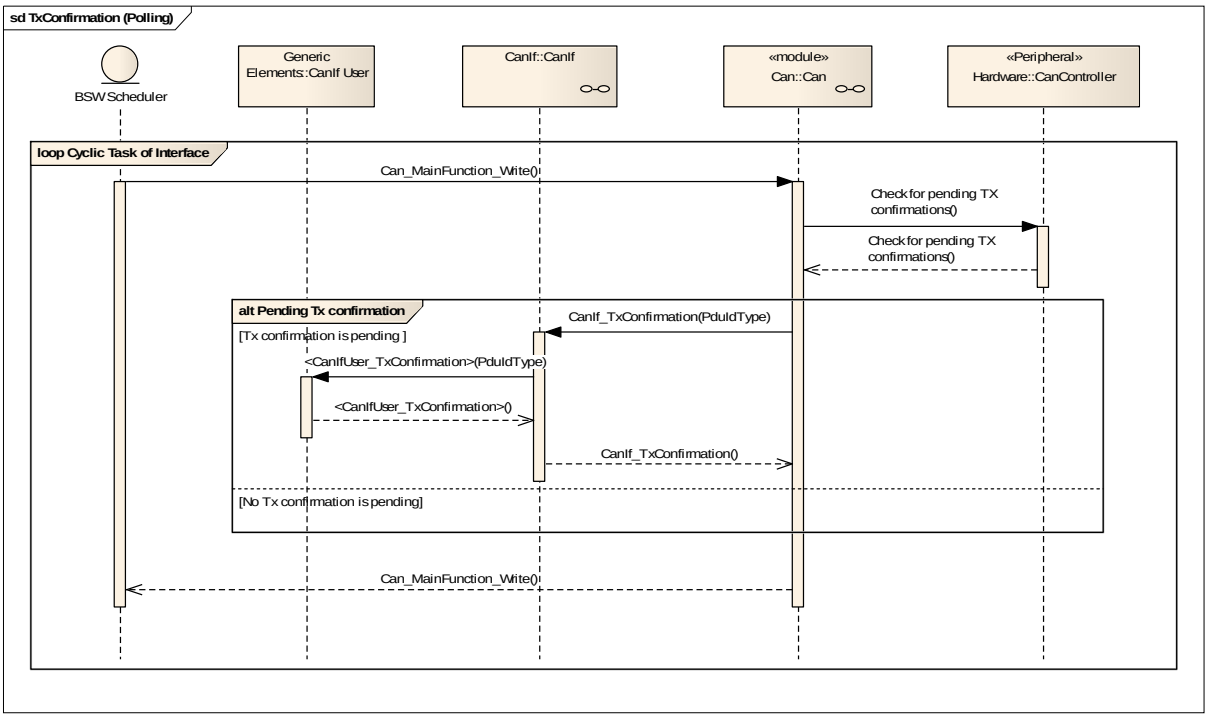

Figure 26 Transmit confirmation polling driven

Activity	***Description***
Cyclic Task CAN Driver	The service `Can_MainFunction_Write()` is called by the BSW Scheduler.
Check for pending transmit confirmations	`Can_MainFunction_Write()` checks the underlying CAN controller(s) about pending transmit confirmations of previously succeeded transmit events.
Transmit Confirmation	The acknowledged CAN frame signals a successful transmission to the sending CAN controller.
Confirmation to CAN Interface	CAN Driver calls the service `CanIf_TxConfirmation()` The parameter CanTxPduId specifies the CAN L-PDU previously sent by `Can_Write()`. The CAN diver must store the all in HTHs pending L-PDU Ids in an array organized per HTH to avoid new search of the L-PDU ID for call of `CanIf_TxConfirmation()`.
Confirmation to upper layer	Calling of the corresponding upper layer confirmation service `<User_TxConfirmation>()`. It signals a successful L-PDU transmission to the upper layer.

9.5 Transmit confirmation (with buffering)

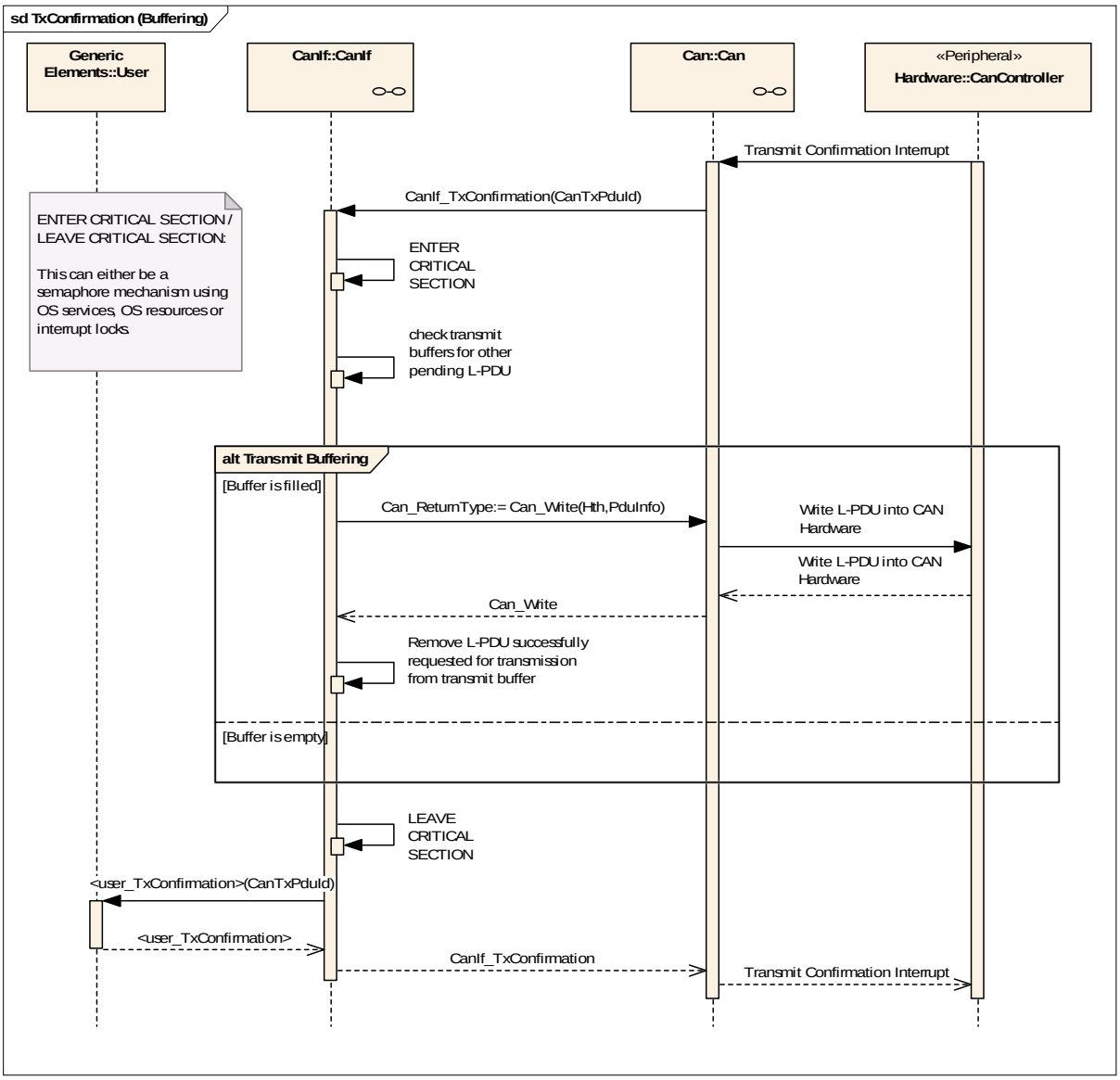

Figure 27 Transmit confirmation with buffering

Activity	*Description*
Transmit interrupt	The acknowledged CAN frame signals a successful transmission to the receiving CAN controller and triggers the transmit interrupt.
Confirmation to CAN Interface	CAN Driver calls the service `CanIf_TxConfirmation()`. The parameter CanTxPduId specifies the CAN L-PDU previously transmitted by `Can_Write()`. The CAN diver must store the all in HTHs pending L-PDU Ids in an array organized per HTH to avoid new search of the L-PDU ID for call of `CanIf_TxConfirmation()`.
ENTER CRITICAL SESSION	Protect transmit buffers from being corrupted. This can be done using interrupt locks, OS resources or semaphores.
Check of transmit buffers	The transmit buffers of the CAN Interface checked, whether a pending L-PDU is stored or not.
Transmit request passed to the CAN Driver	In case of pending L-PDUs in the transmit buffers the highest priority order the latest L-PDU is requested for transmission by `Can_Write()`. It signals a successful L-PDU transmission to the upper layer. Thus `Can_Write()` can be called re-entrant.
Remove transmitted L-PDU from transmit buffers	The L-PDU pending for transmission is removed from the transmission buffers by the CAN Interface.
LEAVE CRITICAL SESSION	End of protection segment.
Confirmation to the upper layer	Calling of the corresponding upper layer confirmation service `<User_TxConfirmation>()`. It signals a successful L-PDU transmission to the upper layer.

9.6 Transmit cancellation (with buffering)

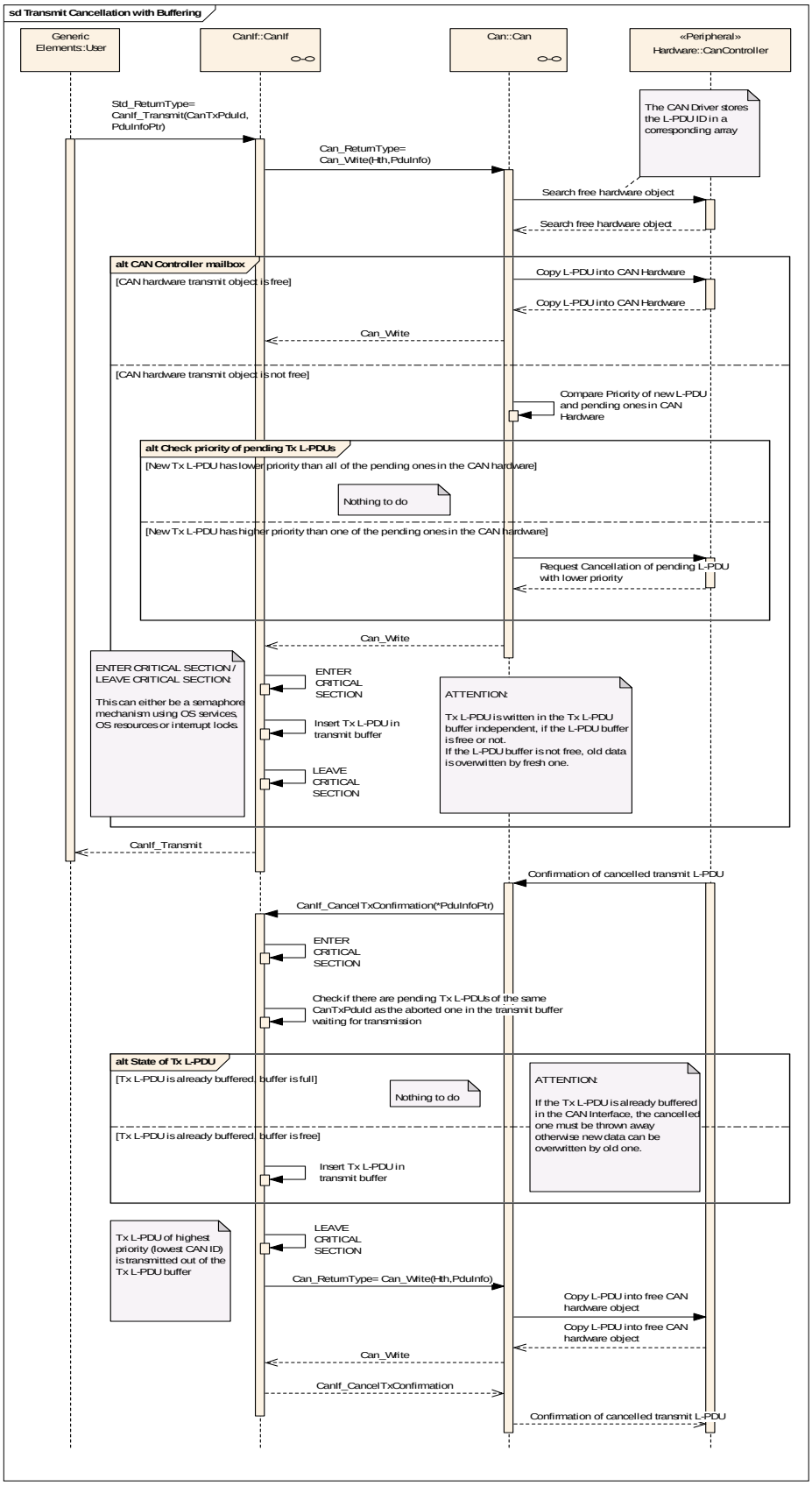

Figure 28 Transmit cancellation

Activity	*Description*
Transmission request	The upper layer initiates a transmit request via the service

	`CanIf_Transmit()`. The parameter CanTxPduId identifies the requested L-PDU. The service performs following steps: - validation of the input parameter - definition of the CAN controller to be used The second parameter *PduInfoPtr is a pointer on the structure with transmit L-PDU related data such as CanSduLength and *CanSduPtr.
Start transmission	`CanIf_Transmit()` requests a transmission and calls the CAN Driver service `Can_Write()` with corresponding processing of the HTH.
Hardware request	`Can_Write()`writes all L-PDU data in the CAN Hardware (if it is free) and sets the hardware request for transmission.
E_OK from Can_Write service	`Can_Write()` returns E_OK to `CanIf_Transmit()`.
E_BUSY from Can_Write service without transmit abort	If the CAN Driver 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, it returns `CAN_E_BUSY` to the CAN Interface.
E_BUSY from Can_Write service with transmit abort	If the CAN Driver detects, there are no free hardware objects available and the new transmit L-PDU has higher priority than all of the pending ones in the CAN hardware, it requested transmit abort of the pending L-PDU in the CAN hardware with the lowest priority and returns `CAN_E_BUSY` to the CAN Interface.
Transmit buffer	The CAN Interface stores the rejected L-PDU in the transmit buffers.
E_OK from CAN Interface	`CanIf_Transmit()` returns E_OK to the upper layer.

Cancellation confirmation notification:

Activity	***Description***
Transmit cancellation confirmation interrupt	The CAN controller signals a successful aborted CAN L-PDU. The CAN Driver detects the abort confirmation event either by interrupt or polling.
Confirmation to CAN Interface	CAN Driver calls service `CanIf_CancelTxConfirmation()`. The parameter CanTxPduId specifies the CAN L-PDU successfully aborted by the CAN Driver. The CAN diver must store the all in HTHs pending L-PDU Ids in an array organized per HTH to avoid new search of the L-PDU ID for call of `CanIf_CancelTxConfirmation()`.
ENTER CRITICAL SESSION	Protect transmit buffers from being corrupted. This can be done using interrupt locks, OS resources or semaphores.
Check of transmit buffers	The transmit buffers of the CAN Interface checked, whether a pending L-PDU of the same CanTxPduId is stored or not. If yes, the cancelled L-PDU is lost. If not, the cancelled L-PDU is stored in the transmit buffer.
Transmit request passed to the CAN Driver	Pending L-PDUs in the transmit buffers with the highest priority order is requested for transmission by `Can_Write()`. It signals a successful L-PDU transmission to the upper layer. Thus `Can_Write()` calls can occur re-entrant.
Remove transmitted L-PDU from transmit buffers	The L-PDU pending for transmission is removed from the transmission buffers by the CAN Interface.
LEAVE CRITICAL SESSION	End of protection segment.
Cancellation confirmation finished	The cancellation confirmation call-out returns.

9.7 Receive indication (interrupt mode)

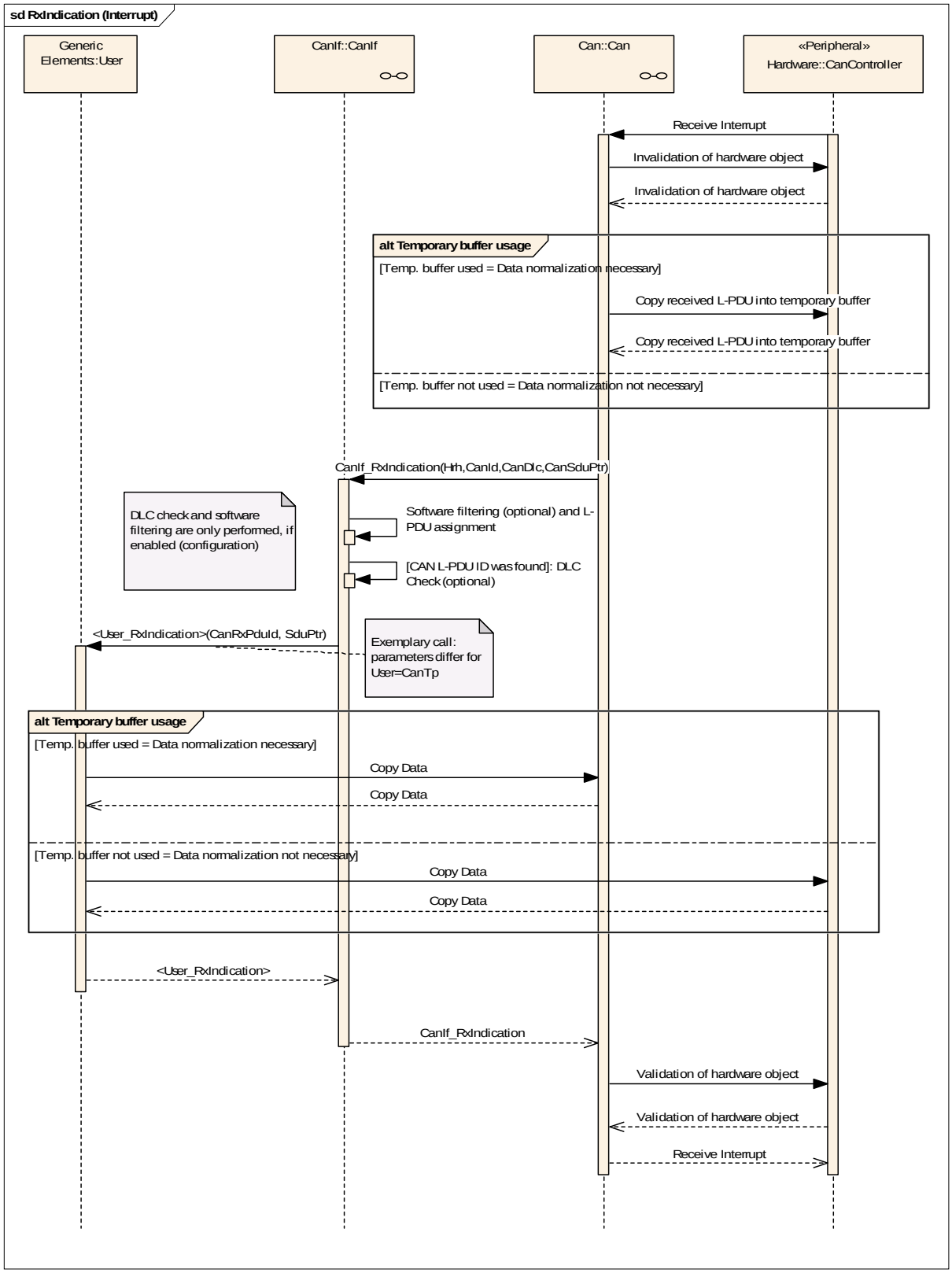

Figure 29 Receive indication interrupt driven

Activity	*Description*
Receive Interrupt	The CAN controller signals a successful reception and triggers a receive interrupt.
Invalidation of CAN hardware object, provide CPU access to CAN mailbox	The CPU (CAN Driver) get exclusive access rights to the CAN mailbox or at least to the corresponding hardware object, where new data were received.
Buffering, normalizing	The L-SDU is normalized and is buffered in the temporary buffer located in the CAN Driver. Each CAN Driver owns a temporary buffer for every physical channel only if normalizing of the data is necessary.
Indication to CAN Interface	The reception is indicated to the CAN Interface by calling of `CanIf_RxIndication()`. The HRH specifies the CAN RAM hardware object and the corresponding CAN controller, which contains the received L-PDU. The temporary buffer is referenced to the CAN Interface by *CanSduPtr.
Software Filtering	The Software Filtering checks, whether the received L-PDU will be processed on a local ECU. If not, the received L-PDU is not indicated to upper layers. Further processing is suppressed.
DLC check	If the L-PDU is found, the DLC of the received L-PDU is compared with the expected, statically configured one for the received L-PDU.
Receive Indication to the upper layer	The corresponding receive indication service of the upper layer is called. This signals a successful reception to the target upper layer. The parameter CanPduId specifies the L-PDU, the second parameter is the reference on the temporary buffer within the L-SDU. During is execution of this service the CAN hardware buffers must be unlocked for CPU access/locked for CAN controller access.
Validation of CAN hardware object, allow access of CAN controller to CAN mailbox	The CAN controller get back exclusive access rights to the CAN mailbox or at least to the corresponding hardware object, where new data were already being copied into the upper layer buffer.

9.8 Receive indication (polling mode)

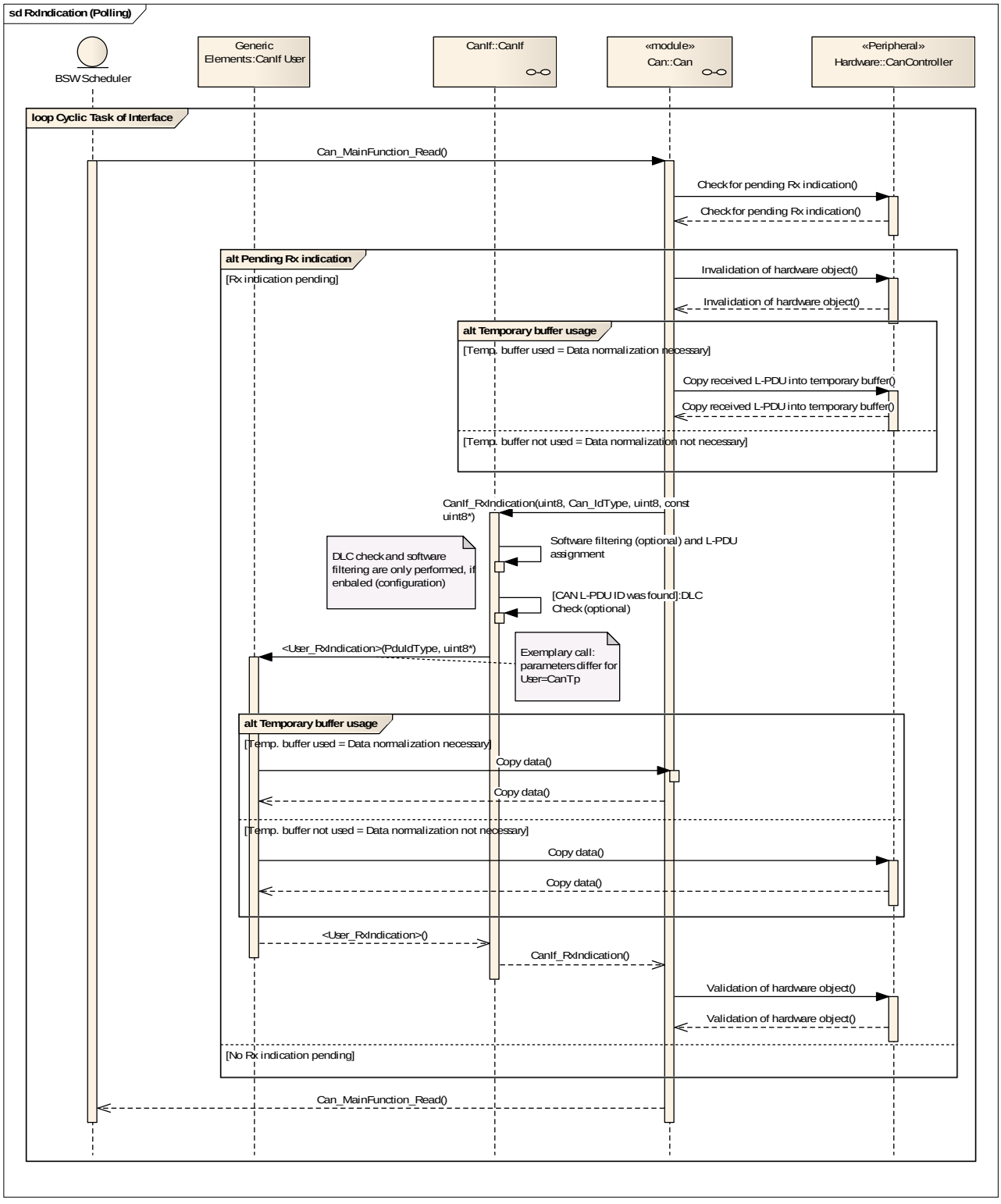

Figure 30 Receive indication polling driven

Activity	*Description*
Cyclic Task **CAN Driver**	The service `Can_MainFunction_Read()` is called by the BSW Scheduler.
Check for new received L-PDU	`Can_MainFunction_Read()` checks the underlying CAN controller(s) about new received L-PDUs.

Invalidation of CAN hardware object, provide CPU access to CAN mailbox	In case of a new receive event the CPU (CAN Driver) get exclusive access rights to the CAN mailbox or at least to the corresponding hardware object, where new data were received.
Buffering, normalizing	In case of a new receive event the L-SDU is normalized and is buffered in the temporary buffer located in the CAN Driver. Each CAN Driver owns such a temporary buffer for every physical channel only if normalizing of the data is necessary.
Indication to CAN Interface	The reception is indicated to the CAN Interface by calling of `CanIf_RxIndication()`. The HRH specifies the CAN RAM hardware object and the corresponding CAN controller, which contains the received L-PDU. The temporary buffer is referenced to the CAN Interface by *CanSduPtr.
Software Filtering	The Software Filtering checks, whether the received L-PDU will be processed on a local ECU. If not, the received L-PDU is not indicated to upper layers. Further processing is suppressed.
DLC check	If the L-PDU is found, the DLC of the received L-PDU is compared with the expected, statically configured one for the received L-PDU.
Receive Indication to the upper layer	If configured, the corresponding receive indication service of the upper layer is called. This signals a successful reception to the target upper layer. The parameter CanPduId specifies the L-PDU, the second parameter is the reference on the temporary buffer within the L-SDU. During is execution of this service the CAN hardware buffers must be unlocked for CPU access/locked for CAN controller access.
Validation of CAN hardware object, allow access of CAN controller to CAN mailbox	The CAN controller get back exclusive access rights to the CAN mailbox or at least to the corresponding hardware object, where new data were already being copied into the upper layer buffer.

9.9 Read received data

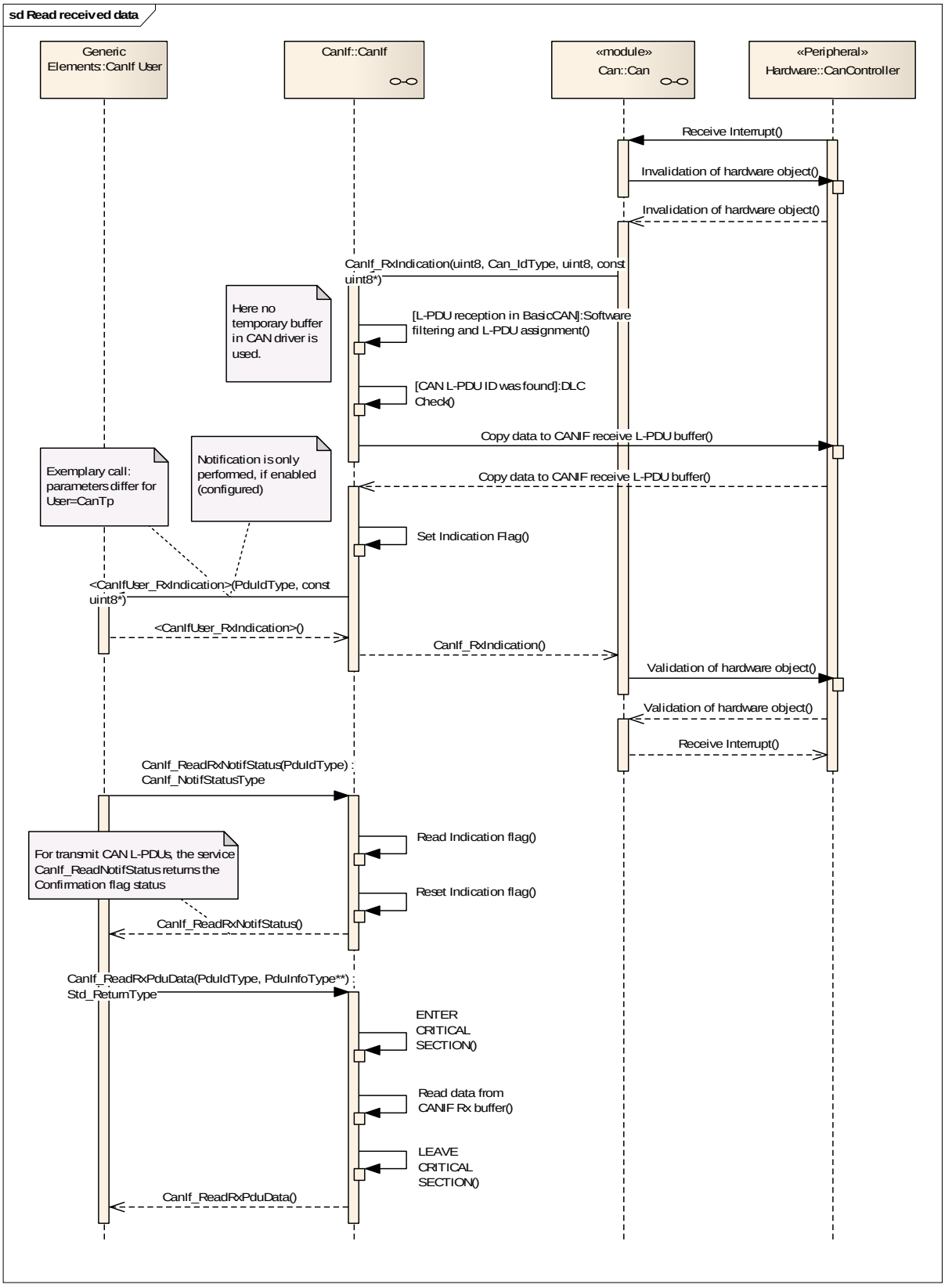

Figure 31 Read received data

Activity	***Description***
Receive Interrupt	The CAN controller signals a successful reception and triggers a receive interrupt.

Invalidation of CAN hardware object, provide CPU access to CAN mailbox	The CPU (CAN Driver) get exclusive access rights to the CAN mailbox or at least to the corresponding hardware object, where new data were received.
Buffering, normalizing	The L-SDU is normalized and is buffered in the temporary buffer located in the CAN Driver. Each CAN Driver owns a temporary buffer for every physical channel only if normalizing of the data is necessary.
Indication to CAN Interface	The reception is indicated to the CAN Interface by calling of `CanIf_RxIndication()`. The HRH specifies the CAN RAM hardware object and the corresponding CAN controller, which contains the received L-PDU. The temporary buffer is referenced to the CAN Interface by *CanSduPtr.
Software Filtering	The Software Filtering checks, whether the received L-PDU will be processed on a local ECU. If not, the received L-PDU is not indicated to upper layers. Further processing is suppressed.
DLC check	If the L-PDU is found, the DLC of the received L-PDU is compared with the expected, statically configured one for the received L-PDU.
Copy data	The data is copied out of the CAN hardware into the receive CAN L-PDU buffers in the CAN Interface. During access the CAN hardware buffers must be unlocked for CPU access/locked fro CAN controller access.
Indication Flag	Set indication status flag for the received L-PDU in the CAN Interface.
Receive Indication to the upper layer	The corresponding receive indication service of the upper layer is called. This signals a successful reception to the target upper layer. The parameter CanPduId specifies the L-PDU, the second parameter is the reference on the temporary buffer within the L-SDU.
Validation of CAN hardware object, allow access of CAN controller to CAN mailbox	The CAN controller get back exclusive access rights to the CAN mailbox or at least to the corresponding hardware object, where new data were already being copied into the upper layer buffer.
Read indication status	Times later the upper layer can read the indication status by call of `CanIf_ReadRxNotifStatus()`. This service can also be used for transmit L-PDUs. Then it return the confirmation status.
Reset indication status	Before `CanIf_ReadRxNotifStatus()` returns, the indication status is reset.
Read received data	Times later the upper layer can read the received data by call of `CanIf_ReadRxNotifStatus()`.

9.10 Start CAN network

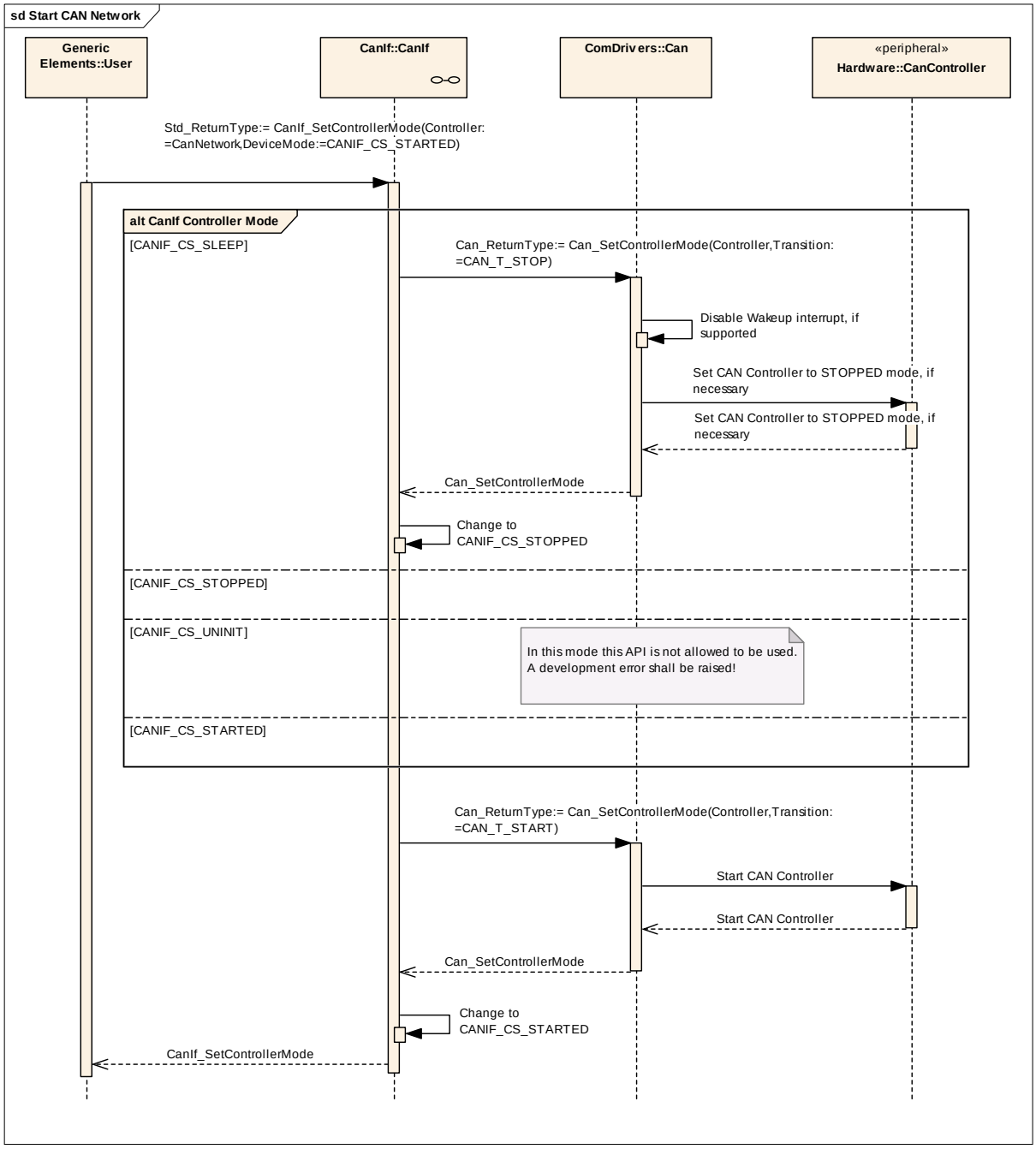

Figure 32 Start CAN network

Activity	***Description***
The upper layer initiates STARTED of the desired CAN controller	The upper layer calls `CanIf_SetControllerMode (Controller, CANIF_CS_STARTED)` to request STARTED mode for the requested CAN network.
CAN Interface checks current operation mode	The CAN Interface determines the current operation mode for the corresponding requested CAN controller(s).
Operation mode is SLEEP	In case of SLEEP the CAN Driver is requested for STOPPED mode by call of `Can_SetControllerMode (Controller, CAN_T_STOPPED)`.
CAN controller is set to STOPPED	CAN controller is requested for STOPPED mode.
CAN Interface is set to STOPPED	The CAN Interface's state machine changes to STOPPED state.
Operation mode is UNINIT	In case of UNINIT an development error shall be raised! In this mode this API is not allowed to be used!

CAN controller is requested for STARTED mode transition	The CAN Interface requests the CAN Driver to initiate a transition to STARTED by `Can_SetControllerMode (Controller, CAN_T_STARTED)`.
CAN controller is set to STARTED mode	The CAN Driver requests CAN controller for STARTED mode.
CAN Driver checks if transition was successful	Inside this call the CAN Driver remains until either the sleep transition was successful or the hardware loop timer elapses.
CAN Interface's corresponding CAN controller is set to STARTED	After successful transition the CAN Interface changes the corresponding CAN controller mode to STARTED mode.

9.11 Stop & sleep CAN network

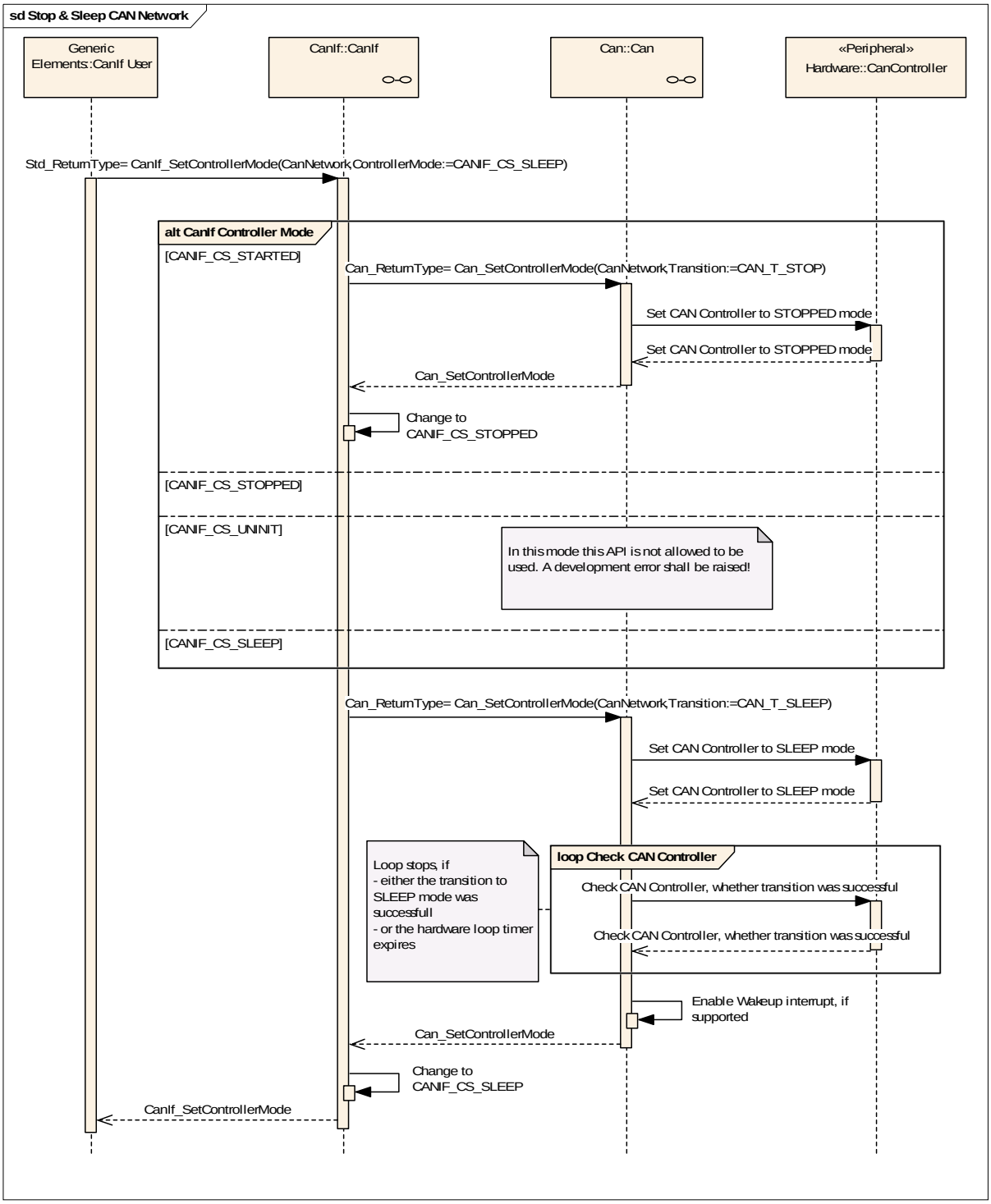

Figure 33 Stop CAN network

Activity	*Description*
The upper layer initiates SLEEP of the desired CAN controller	The upper layer calls `CanIf_SetControllerMode (Controller, CANIF_CS_SLEEP)` to request SLEEP mode for the requested CAN network.
CAN Interface checks current operation mode	The CAN Interface determines the current operation mode for the corresponding requested CAN network.
Operation mode is STARTED	In case of STARTED the CAN Driver is requested for STOPPED mode by call of `Can_SetControllerMode (Controller, CAN_T_STOPPED)`.
CAN controller is set to STOPPED	CAN controller is requested for STOPPED mode.
CAN Interface is set to STOPPED	The CAN Interface's state machine changes to STOPPED state.
Operation mode is UNINIT	In case of UNINIT an development error shall be raised! In this mode this API is not allowed to be used!
CAN Driver is requested to initiate SLEEP mode transition	The CAN Interface requests the CAN Driver to initiate SLEEP transition by `Can_SetControllerMode (Controller, CAN_T_SLEEP)`.
CAN controller is set to SLEEP mode	The CAN Driver sets CAN controller in SLEEP mode.
CAN Driver checks if transition was successful	Inside this call the CAN Driver stays until either the sleep transition was successful or the hardware loop timer elapses.
Wakeup interrupt is enabled	After successful transition to SLEEP the CAN Driver enables the wakeup interrupt, if provided by the CAN Driver.
CAN Interface changes to SLEEP mode	The CAN Interface's state machine for the requested CAN controller changes to SLEEP state.

9.12 BusOff notification

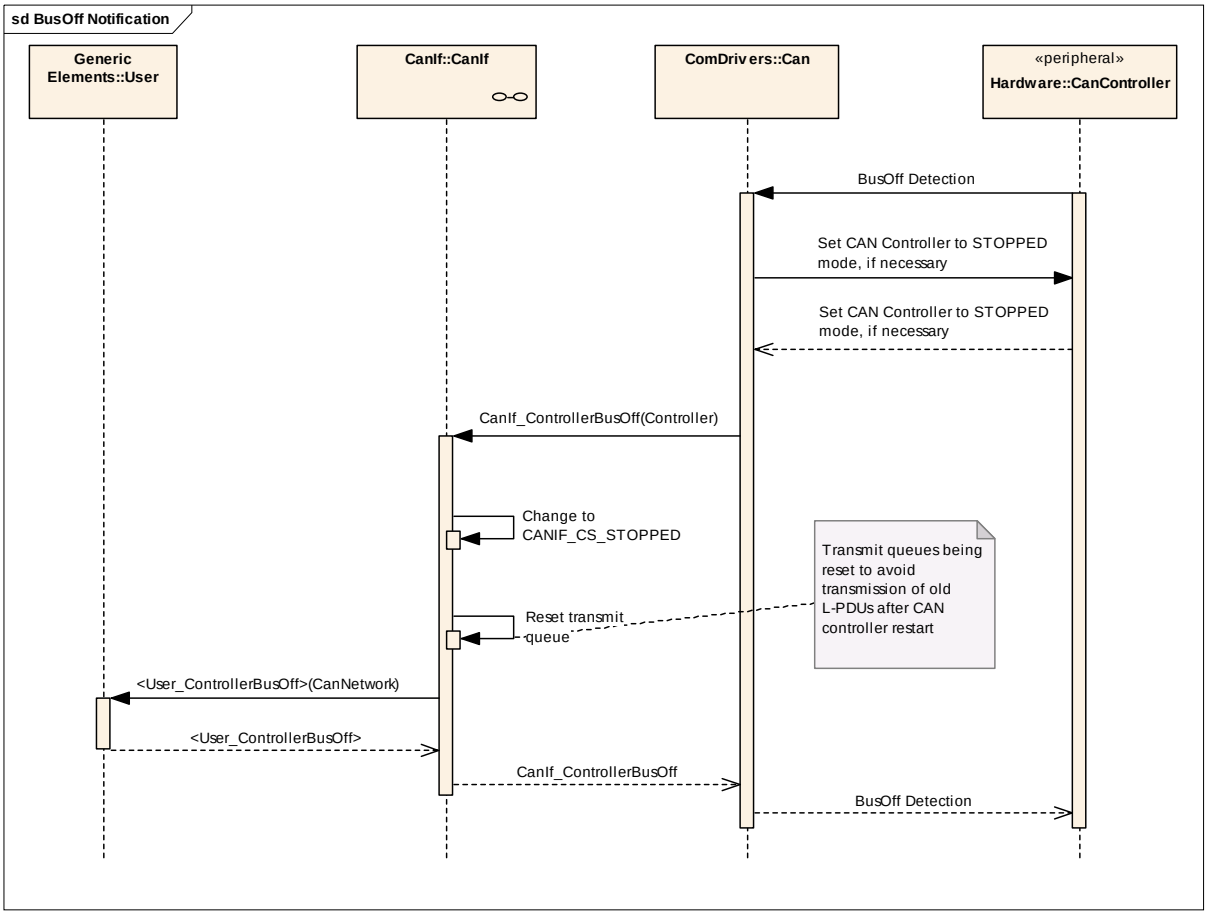

Figure 34 BusOff notification

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 CAN Interface 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

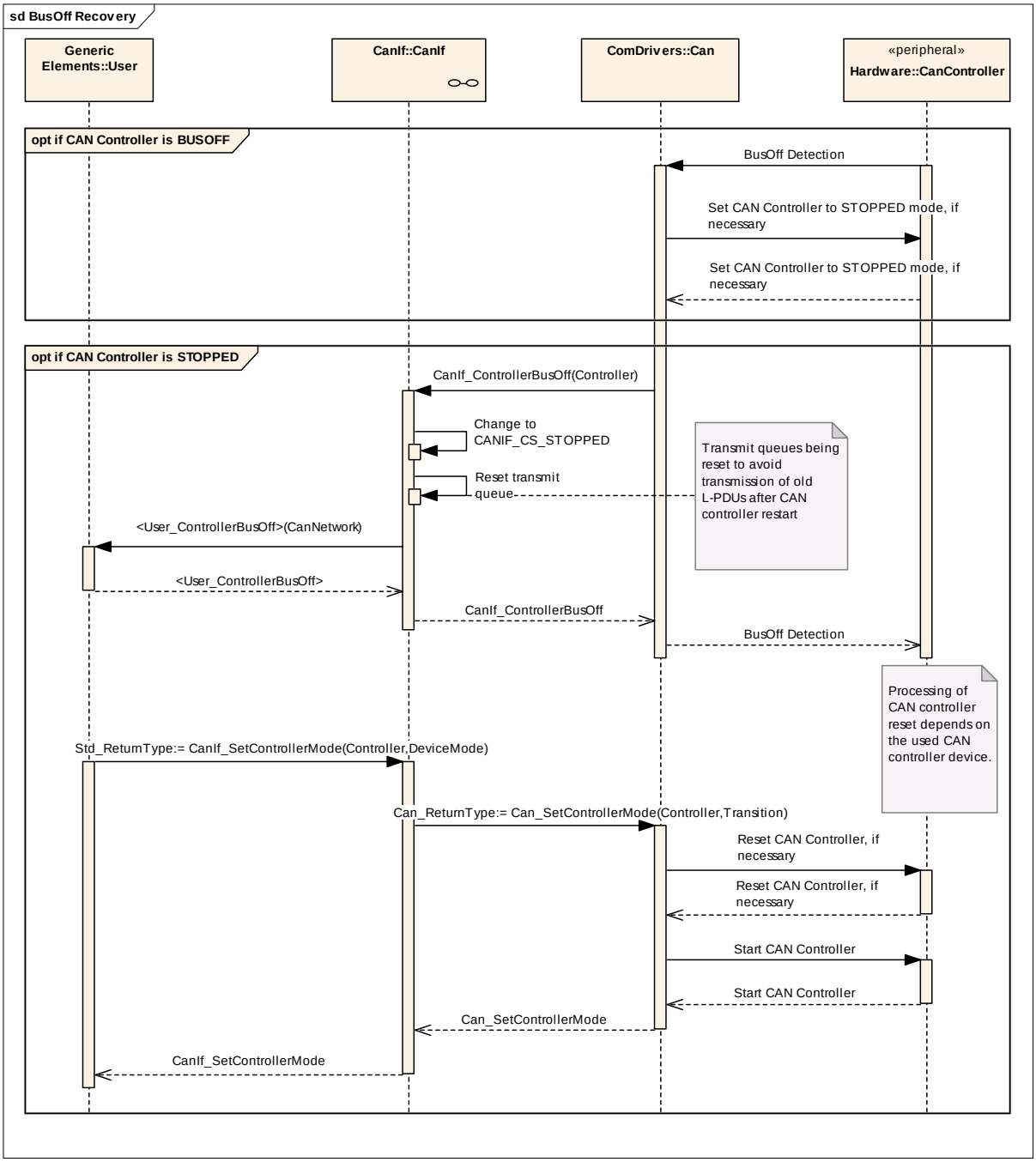

Figure 35 BusOff recovery

	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 CAN Interface by calling of `CanIf_ControllerBusOff()`.The transmit buffers inside the CAN Interface 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 (Controller, CANIF_CS_STARTED)`.
Reset of CAN controller	If necessary the CAN Driver resets the CAN controller by `Can_InitController(Controller, ConfigurationIndex)`.
Restart of CAN controller	The driver restarts the CAN controller by call of `Can_SetControllerMode (Controller, CAN_T_STARTED)`.

10 Configuration specification

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

Chapter 10.2 specifies the structure (containers) and the parameters of the module CAN Interface.

Chapter 10.3 specifies published information of the module CAN Interface.

10.1 How to read this chapter

In addition to this section, it is highly recommended to read the documents:

- [2] Layered Software Architecture
- [6] Specification of ECU Configuration
  This document describes the AUTOSAR configuration methodology and the AUTOSAR configuration meta model in detail.

The following is only a short survey of the topic and it will not replace the ECU Configuration Specification document.

10.1.1 Configuration and configuration parameters

Configuration parameters define the variability of the generic part(s) of an implementation of a module. This means that only generic or configurable module implementation can be adapted to the environment (software/hardware) in use during system and/or ECU configuration.

The configuration of parameters can be achieved at different times during the software process: before compile time, before link time or after build time. In the following, the term “configuration class” (of a parameter) shall be used in order to refer to a specific configuration point in time.

10.1.2 Variants

Variants describe sets of configuration parameters. E.g., variant 1: only pre-compile time configuration parameters; variant 2: mix of pre-compile- and post build time-configuration parameters. In one variant a parameter can only be of one configuration class.

10.1.3 Containers

Containers structure the set of configuration parameters. This means:

- *all* configuration parameters are kept in containers.

- (sub-) containers can reference (sub-) containers. It is possible to assign a multiplicity to these references. The multiplicity then defines the possible number of instances of the contained parameters.

10.1.4 Specification template for configuration parameters

The following tables consist of three sections:
- the general section
- the configuration parameter section
- the section of included/referenced containers

Pre-compile time - specifies whether the configuration parameter shall be of configuration class *Pre-compile time* or not

Label	***Description***
x	The configuration parameter shall be of configuration class *Pre-compile time*.
--	The configuration parameter shall never be of configuration class *Pre-compile time*.

Link time - specifies whether the configuration parameter shall be of configuration class *Link time* or not

Label	***Description***
x	The configuration parameter shall be of configuration class *Link time*.
--	The configuration parameter shall never be of configuration class *Link time*.

Post Build - specifies whether the configuration parameter shall be of configuration class *Post Build* or not

Label	***Description***
x	The configuration parameter shall be of configuration class *Post Build* and no specific implementation is required.
L	*Loadable* - the configuration parameter shall be of configuration class *Post Build* and only one configuration parameter set resides in the ECU.
M	*Multiple* - the configuration parameter shall be of configuration class *Post Build* and is selected out of a set of multiple parameters by passing a dedicated pointer to the init function of the module.
--	The configuration parameter shall never be of configuration class *Post Build*.

10.2 Containers and configuration parameters

The following chapters summarize all configuration parameters. The detailed meanings of the parameters describe chapter [7 Functional specification] and chapter [8 API specification].

CANIF104: The listed configuration items can be derived from a network description database, which is based on the EcuConfigurationTemplate. The configuration tool shall extract all information to configure the CAN Interface.

CANIF131: The consistency of the configuration must be checked by the configuration tool at configuration time. Configuration rules and constraints for plausibility checks shall be performed during configuration time, where possible.

CANIF066: The CAN Interface has access to the CAN Driver configuration data. All public CAN Driver configuration data are described in [8] Specification of CAN Driver.

CANIF132: These dependencies between CAN Driver and CAN Interface configuration must be provided at configuration time by the configuration tools.

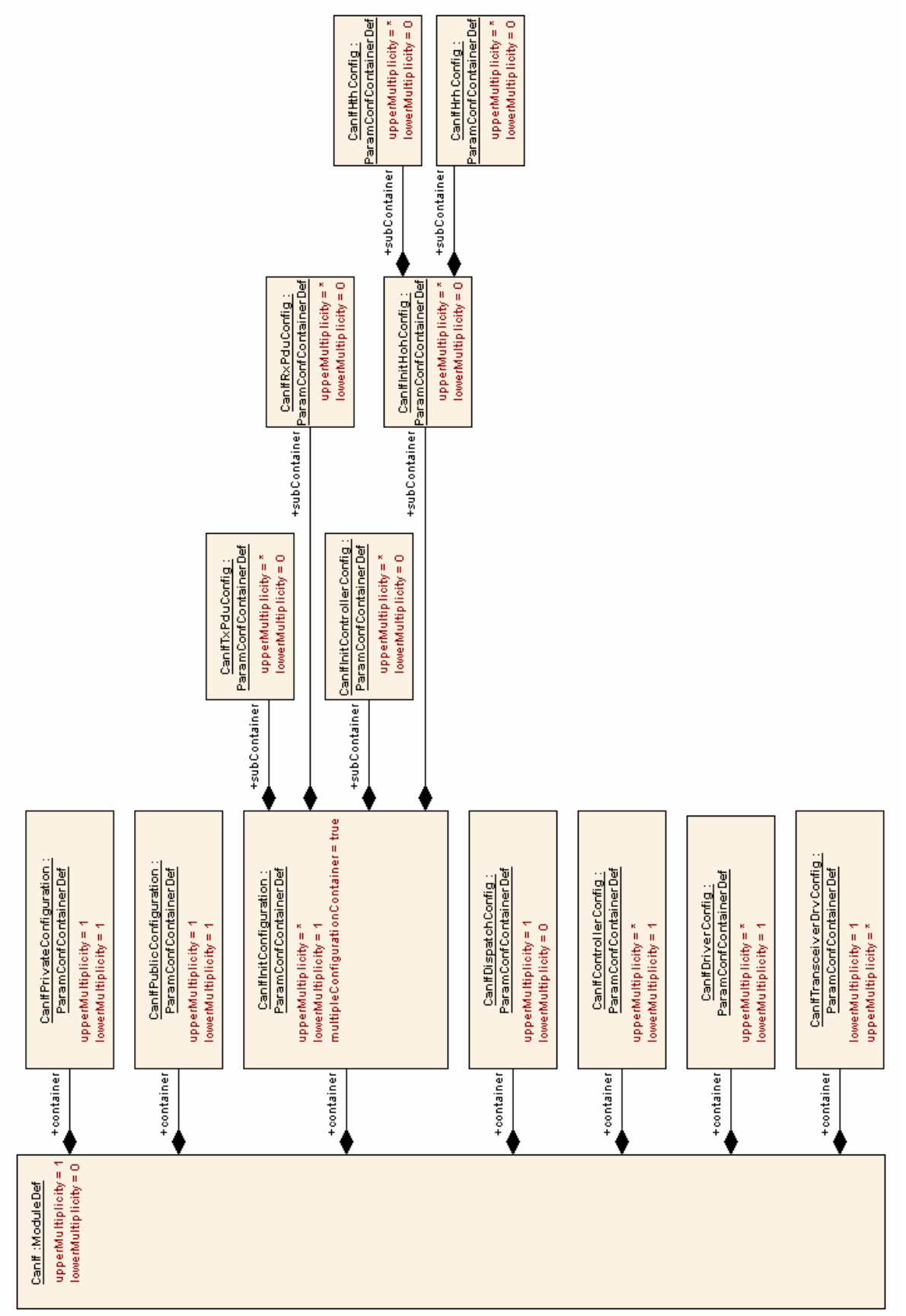

Figure 36 Overview about CAN Interface configuration containers

10.2.1 Variants

CANIF228:
VARIANT-PRE-COMPILE: Only pre compile time parameters.
VARIANT-LINK-TIME: Mix of pre compile- and link time parameters.
VARIANT-POST-BUILD: Mix of pre compile-, link time and post build time parameters.

CANIF240: For post build time parameters the type “x” was chosen to allow both variants of implementations with either loadable (“L”) or multiple (“M”) types of post built parameters.

10.2.2 CanIf

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***
CanIfControllerConfig	1..*	This container contains the configuration (parameters) of all addressed CAN controllers by each underlying CAN driver.
CanIfDispatchConfig	0..1	Callout functions with respect to the upper layers. This callout functions defined in this container are common to all configured underlying CAN Drivers / CAN Transceiver Drivers.
CanIfDriverConfig	1..*	Configuration parameters for all the underlying CAN drivers are aggregated under this container.
CanIfInitConfiguration	1..*	This container contains the init parameters of the CAN Interface.
CanIfPrivateConfiguration	1	This container contains the private configuration (parameters) of the CAN Interface.
CanIfPublicConfiguration	1	This container contains the public configuration (parameters) of the CAN Interface.
CanIfTransceiverDrvConfig	1..*	This container contains the configuration (parameters) of all addressed CAN transceivers by each underlying CAN Transceiver Driver.

10.2.3 CanIfPrivateConfiguration

SWS Item	**CANIF245 :**
Container Name	CanIfPrivateConfiguration{CanInterfacePrivateConfiguration }
Description	This container contains the private configuration (parameters) of the CAN Interface.
Configuration Parameters	

SWS Item	**--**		
Name	CanIfDlcCheck {CANIF_DLC_CHECK}		
Description	Selects whether the DLC check is supported. True: Enabled False: Disabled		
Multiplicity	1		
Type	BooleanParamDef		
Default value	true		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	

Scope / Dependency	scope: Module

SWS Item	**--**		
Name	CanIfNumberOfTxBuffers {CANIF_NUMBER_OF_TXBUFFERS}		
Description	Defined the number of L-PDU elements for the transmit buffering. The Tx L-PDU buffers shall be used to store an L-PDU once for each different L-PDU handle. Range: 0..max. number of Tx L-PDUs to be used. Default Value: NUMBER_OF_TX_PDUS		
Multiplicity	1		
Type	IntegerParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Module		

SWS Item	**--**		
Name	CanIfSoftwareFilterType {CANIF_SOFTWARE_FILTER_TYPE}		
Description	Selects the desired software filter mechanism for reception only. Each implemented software filtering method is identified by this enumeration number. Range: Types implemented software filtering methods		
Multiplicity	1		
Type	EnumerationParamDef		
Range	BINARY	Selects Binary Filter method.	
	INDEX	Selects Index Filter method.	
	LINEAR	Selects Linear Filter method.	
	TABLE	Selects Table Filter method.	
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Module dependency: BasicCAN reception must be enabled by CANIF_HRH_TYPE for at least one HRH.		

No Included Containers

10.2.4 CanIfPublicConfiguration

SWS Item **CANIF246 :**
Container Name CanIfPublicConfiguration{CanInterfacePublicConfiguration }
Description This container contains the public configuration (parameters) of the CAN Interface.
Configuration Parameters

SWS Item	**--**		
Name	CanIfDevErrorDetect {CANIF_DEV_ERROR_DETECT}		
Description	Enables and disables the development error detection and notification mechanism. True: Enabled False: Disabled		
Multiplicity	1		
Type	BooleanParamDef		
Default value	true		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Module		

SWS Item	**--**		
Name	CanIfMultipleDriverSupport {CANIF_MULTIPLE_DRIVER_SUPPORT}		
Description	Selects support for multiple CAN Drivers. True: Enabled False: Disabled		
Multiplicity	1		
Type	BooleanParamDef		
Default value	true		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfNumberOfCanHwUnits {CANIF_NUMBER_OF_CAN_HW_UNITS}		
Description	Number of served CAN hardware units. Range: 1..max. number of underlying supported CAN Hardware units		
Multiplicity	1		
Type	IntegerParamDef		
Default value	1		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfReadRxPduDataApi {CANIF_READRXPDU_DATA_API}		
Description	Enables / Disables the API CanIf_ReadRxPduData() for reading received L-PDU data. True: Enabled False: Disabled		
Multiplicity	1		
Type	BooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfReadRxPduNotifyStatusApi {CANIF_READRXPDU_NOTIF_STATUS_API}		
Description	Enables and disables the API for reading the received L-PDU data. True: Enabled False: Disabled		
Multiplicity	1		
Type	BooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**--**
Name	CanIfReadTxPduNotifyStatusApi {CANIF_READTXPDU_NOTIF_STATUS_API}
Description	Enables and disables the API for reading the notification status of transmit and receive L-PDUs. True: Enabled False: Disabled

Multiplicity	1		
Type	BooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfSetDynamicTxIdApi {CANIF_SETDYNAMICTXID_API}		
Description	Enables and disables the API for reconfiguration of the CAN Identifier for each Transmit L-PDU. True: Enabled False: Disabled		
Multiplicity	1		
Type	BooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfVersionInfoApi {CANIF_VERSION_INFO_API}		
Description	Enables and disables the API for reading the version information about the CAN Interface. True: Enabled False: Disabled		
Multiplicity	1		
Type	BooleanParamDef		
Default value	true		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**--**		
Name	CanIfWakeupEventApi {CANIF_WAKEUP_EVENT_API}		
Description	Enables and disables the API for wakeup support called by the CAN Driver. True: Enabled False: Disabled		
Multiplicity	1		
Type	BooleanParamDef		
Default value	true		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Wakeup support must be enabled by the CAN Driver configuration.		

No Included Containers

10.2.5 CanIfInitConfiguration

SWS Item	**CANIF247 :**
Container Name	CanIfInitConfiguration{CanInterfaceInitConfiguration} [Multi Config Container]
Description	This container contains the init parameters of the CAN Interface.
Configuration Parameters	

SWS Item	**--**		
Name	CanIfConfigSet {CANIF_CONFIGSET}		
Description	Selects the CAN Interface specific configuration setup. This type of the external data structure shall contain the post build initialization data for the CAN Interface for all underlying CAN Dirvers. constant to CanIf_ConfigType		
Multiplicity	1		
Type	StringParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Module		

SWS Item	**--**		
Name	CanIfNumberOfCanRxPduIds {CANIF_NUMBER_OF_CANRXPDUIDS}		
Description	Total number of CanRxPduIds to be handled. Range: 0..max number of defined CanRxPduIds		
Multiplicity	1		
Type	IntegerParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfNumberOfCanTXPduIds {CANIF_NUMBER_OF_CANTXPDUIDS}		
Description	Total number of CanTxPduIds to be handled. Range: 0..max number of defined CanTxPduIds		
Multiplicity	1		
Type	IntegerParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfNumberOfDynamicCanTXPduIds {CANIF_NUMBER_OF_DYNAMIC_CANTXPDUIDS}		
Description	Total number of dynamic CanTxPduIds to be handled. Range: 0..max. nember of defined CanTxPduIds		
Multiplicity	1		
Type	IntegerParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanIfInitControllerConfig	0..*	This container contains the references to the configuration

		setup of each underlying CAN driver.
CanIfInitHohConfig	0..*	This container contains the references to the configuration setup of each underlying CAN Driver.
CanIfRxPduConfig	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.
CanIfTxPduConfig	0..*	This container contains the configuration (parameters) of each transmit CAN L-PDU. The SHORT-NAME of "CanIfTxPduConfig" container represents the symolic name of Transmit L-PDU.

10.2.6 CanIfTxPduConfig

SWS Item	**CANIF248 :**
Container Name	CanIfTxPduConfig{CanInterfaceTxPduConfiguration}
Description	This container contains the configuration (parameters) of each transmit CAN L-PDU. The SHORT-NAME of "CanIfTxPduConfig" container represents the symolic name of Transmit L-PDU.
Configuration Parameters	

SWS Item	**--**		
Name	CanIfCanTxPduId {CANIF_CANTXPDUID}		
Description	ECU wide unique, symbolic handle for transmit CAN L-PDU. The CanIfCanTxPduId is configurable at pre-compile and post-built time. Range: 0..max. number of CantTxPduIds		
Multiplicity	1		
Type	IntegerParamDef (Symbolic Name generated for this parameter)		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfCanTxPduIdCanId {CANIF_CANTXPDUID_CANID}		
Description	CAN Identifier of transmit CAN L-PDUs used by the CAN Driver for CAN L-PDU transmission. Range: 11 Bit For Standard CAN Identifier ... 29 Bit For Extended CAN identifier		
Multiplicity	1		
Type	IntegerParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Network		

SWS Item	**--**	
Name	CanIfCanTxPduIdDlc {CANIF_CANTXPDUID_DLC}	
Description	Data length code (in bytes) of transmit CAN L-PDUs used by the CAN Driver for CAN L-PDU transmission. The data area size of a CAN L-Pdu can have a range from 0 to 8 bytes.	
Multiplicity	1	
Type	IntegerParamDef	
Range	0 .. 8	

Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Network dependency: CanIfNumberOfTxBuffers		

SWS Item	**--**		
Name	CanIfCanTxPduType {CANIF_CANTXPDUID_TYPE}		
Description	Defines the type of each transmit CAN L-PDU.		
Multiplicity	1		
Type	EnumerationParamDef		
Range	DYNAMIC	CAN ID is defined at runtime.	
	STATIC	CAN ID is defined at compile-time.	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfReadTxPduNotifyStatus {CANIF_READTXPDU_NOTIFY_STATUS}		
Description	Enables and disables transmit confirmation for each transmit CAN L-PDU for reading its notification status. True: Enabled False: Disabled		
Multiplicity	1		
Type	BooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Module dependency: CANIF_READTXPDU_NOTIFY_STATUS_API must be enabled.		

SWS Item	**--**		
Name	CanIfTxPduIdCanIdType {CANIF_CANIFTXPDUID_CANIDTYPE}		
Description	CAN Identifier of transmit CAN L-PDUs used by the CAN Driver for CAN L-PDU transmission.		
Multiplicity	1		
Type	EnumerationParamDef		
Range	EXTENDED_CAN	The CANID is of type Extended (29 bits)	
	STANDARD_CAN	The CANID is of type Standard (11 bits)	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Network		

SWS Item	**--**	
Name	CanIfTxUserType {CANIF_TX_USER_TYPE}	
Description	This parameter defines the type of the transmit confirmation call-out called to the corresponding upper layer the used TargetTxPduId belongs to.	
Multiplicity	1	
Type	EnumerationParamDef	
Range	CAN_NM	CAN NM
	CAN_TP	CAN TP
	PDUR	PDU Router

ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfUserTxConfirmation {CANIF_USER TX_CONFIRMATION}		
Description	Name of target confirmation services to target upper layers (PduR, CanNm and CanTp. If parameter is not configured then no call-out function is provided by the upper layer for this Tx L-PDU.		
Multiplicity	0..1		
Type	FunctionNameDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfCanTxPduHthRef {CANIF_HTH_REF_ID}		
Description	Handle, that defines the hardware object or the pool of hardware objects configured for transmission. The parameter refers HTH Id, to which the L-PDU belongs to.		
Multiplicity	0..*		
Type	Reference to CanIfHthConfig		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency			

SWS Item	**--**		
Name	PduIdRef		
Description	Reference to the "global" Pdu structure to allow harmonization of handle IDs in the COM-Stack.		
Multiplicity	1		
Type	Reference to Pdu		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	--	
Scope / Dependency			

No Included Containers

10.2.7 CanIfRxPduConfig

SWS Item	**CANIF249 :**
Container Name	CanIfRxPduConfig{CanInterfaceRxPduConfiguration}
Description	This container contains the configuration (parameters) of each receive CAN L-PDU. The SHORT-NAME of "CanIfRxPduConfig" container itself represents the symolic name of Receive L-PDU.
Configuration Parameters	

SWS Item	**--**

Name	CanIfCanRxPduCanId {CANIF_CANRXPDUID_CANID}		
Description	CAN Identifier of Receive CAN L-PDUs used by the CAN Interface. Exa: Software Filtering. Range: 11 Bit For Standard CAN Identifier ... 29 Bit For Extended CAN identifier		
Multiplicity	1		
Type	IntegerParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Network		

SWS Item	**--**		
Name	CanIfCanRxPduDlc {CANIF_CANRXPDUID_DLC}		
Description	Data Length code of received CAN L-PDUs used by the CAN Interface. Exa: DLC check. The data area size of a CAN L-PDU can have a range from 0 to 8 bytes.		
Multiplicity	1		
Type	IntegerParamDef		
Range	0 .. 8		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Network		

SWS Item	**--**		
Name	CanIfCanRxPduId {CANIF_CANRXPDUID}		
Description	ECU wide unique, symbolic handle for receive CAN L-PDU. The CanRxPduId is configurable at pre-compile and post-built time. It shall fulfill ANSI/AUTOSAR definitions for constant defines. Range: 0..max. number of defined CanRxPduIds		
Multiplicity	1		
Type	IntegerParamDef (Symbolic Name generated for this parameter)		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfReadRxPduData {CANIF_READRXPDU_DATA}		
Description	Enables and disables the Rx buffering for reading of received L-PDU data. True: Enabled False: Disabled		
Multiplicity	1		
Type	BooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU dependency: CANIF_CANPDUID_READDATA_API must be enabled.		

SWS Item	**--**
Name	CanIfReadRxPduNotifyStatus {CANIF_READRXPDU_NOTIFY_STATUS}

Description	Enables and disables receive indication for each receive CAN L-PDU for reading its' notification status. True: Enabled False: Disabled		
Multiplicity	1		
Type	BooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Module dependency: CANIF_READRXPDU_NOTIFY_STATUS_API must be enabled.		

SWS Item	**--**		
Name	CanIfRxPduIdCanIdType {CANIF_CANRXPDUID_CANIDTYPE}		
Description	CAN Identtifier of receive CAN L-PDUs used by the CAN Driver for CAN L-PDU reception.		
Multiplicity	1		
Type	EnumerationParamDef		
Range	EXTENDED_CAN	The CANID is of type Extended (29 bits)	
	STANDARD_CAN	The CANID is of type Standard (11 bits)	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Network		

SWS Item	**--**		
Name	CanIfRxUserType {CANIF_RX_USER_TYPE}		
Description	This parameter defines the type of the receive indication call-outs called to the corresponding upper layer the used TargetRxPduId belongs to.		
Multiplicity	1		
Type	EnumerationParamDef		
Range	CAN_NM	CAN NM	
	CAN_TP	CAN TP	
	PDUR	PDU Router	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfUserRxIndication {CANIF_USER_RX_INDICATION}		
Description	Name of target indication services to target upper layers (PduRouter, CanNm, CanTp and ComplexDeviceDrivers). If parameter is 0 no call-out function is configured.		
Multiplicity	1		
Type	FunctionNameDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**--**
Name	CanIfCanRxPduHrhRef {CANIF_HRH_REF_ID}

Description	The HRH to which Rx L-PDU belongs to, is referred through this parameter.		
Multiplicity	0..*		
Type	Reference to CanIfHrhConfig		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Module dependency: This information has to be derived from the CAN Driver configuration.		

SWS Item	**--**		
Name	PduIdRef		
Description	Reference to the "global" Pdu structure to allow harmonization of handle IDs in the COM-Stack.		
Multiplicity	1		
Type	Reference to Pdu		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

No Included Containers

10.2.8 CanIfDispatchConfig

SWS Item	**CANIF250 :**
Container Name	CanIfDispatchConfig{CanInterfaceDispatcherConfiguration }
Description	Callout functions with respect to the upper layers. This callout functions defined in this container are common to all configured underlying CAN Drivers / CAN Transceiver Drivers.
Configuration Parameters	

SWS Item	**--**		
Name	CanIfBusOffNotification {CANIF_USER_BUSOFF_NOTIFICATION}		
Description	Name of target BusOff notification services to target upper layers (PduRouter, CanNm, CanTp and ComplexDeviceDrivers).		
Multiplicity	1		
Type	FunctionNameDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Any notification call-out to upper layers must be configured.		

SWS Item	**--**
Name	CanIfWakeupNotification {CANIF_USER_WAKEUP_NOTIFICATION}
Description	Name of target wakeup notification services to target upper layers (PduRouter, CanNm, CanTp and ComplexDeviceDrivers). If parameter is 0 no call-out function is configured.
Multiplicity	0..1
Type	FunctionNameDef
Default value	--

ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Only if supported by CAN controller and enabled by CAN Driver configuration.		

SWS Item	**--**		
Name	CanIfWakeupValidNotification {CANIF_USER_WAKEUP_VALIDATION_NOTIFICATION}		
Description	Name of target wakeup validation notification services to target upper layers (ECU State Manager). If parameter is 0 no call-out function is configured.		
Multiplicity	0..1		
Type	FunctionNameDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Only if supported by CAN controller and enabled by CAN Driver configuration.		

No Included Containers

10.2.9 CanIfControllerConfig

SWS Item **--**
Container Name CanIfControllerConfig{CanInterfaceControllerConfiguration}
Description This container contains the configuration (parameters) of all addressed CAN controllers by each underlying CAN driver.
Configuration Parameters

SWS Item	**--**		
Name	CanIfWakeupSupport {CANIF_WAKEUP_SUPPORT}		
Description	Enables wakeup support and defines the source device of a wakeup event.		
Multiplicity	1		
Type	EnumerationParamDef		
Range	CONTROLLER	Wakeup by CAN Controller is supported	
	NO_WAKEUP	No wakeup is supported	
	TRANSCEIVER	Wakeup by CAN Transceiver is supported	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: Network		

SWS Item	**--**		
Name	CanIfControllerIdRef		
Description	Logical handle of the underlying CAN controller to be served by the CAN Interface. Range: 0..max. number of underlying supported		
Multiplicity	0..*		
Type	Reference to CanController		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE

	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfDriverNameRef {CANIF_DRIVER_REF_NAME}		
Description	Refers to the CAN Driver Name to which the controller belongs to. This parameter refers to CanIfDriverConfig container.		
Multiplicity	1		
Type	Reference to CanIfDriverConfig		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency			

SWS Item	**--**		
Name	CanIfInitControllerRef		
Description	Reference to the Init Controller Configuration.		
Multiplicity	1		
Type	Reference to CanIfInitControllerConfig		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency			

No Included Containers

10.2.10 CanIfInitControllerConfig

SWS Item	**CANIF252 :**
Container Name	CanIfInitControllerConfig{CanInterfaceInitControllerConfiguration}
Description	This container contains the references to the configuration setup of each underlying CAN driver.
Configuration Parameters	

SWS Item	**--**		
Name	CanIfControllerRefConfigSet {CANIF_CONTROLLER_REF_CONFIGSET}		
Description	References the corresponding CAN Controller configuration setup of the corresponding CAN Driver.		
Multiplicity	1		
Type	Reference to CanController		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency			

No Included Containers

10.2.11 CanIfDriverConfig

SWS Item	**CANIF253 :**

Container Name	CanIfDriverConfig{CanInterfaceDriverConfiguration}
Description	Configuration parameters for all the underlying CAN drivers are aggregated under this container.
Configuration Parameters	

SWS Item	**--**		
Name	CanIfBusoffNotification {CANIF_BUSOFF_NOTIFICATION}		
Description	Selects whether BusOff indication notification is supported. True: Enabled False: Disabled		
Multiplicity	1		
Type	BooleanParamDef		
Default value	true		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfReceiveIndication {CANIF_RECEIVE_INDICATION}		
Description	Selects whether receive indication notification is supported. True: Enabled False: Disabled		
Multiplicity	1		
Type	BooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfTransmitCancellation {CANIF_TRANSMIT_CANCELLATION}		
Description	Selects whether transmit cancellation is supported. True: Enabled False: Disabled		
Multiplicity	1		
Type	BooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Module dependency: CANIF_TRANSMIT_BUFFER must be enabled		

SWS Item	**--**		
Name	CanIfTxConfirmation {CANIF_TRANSMIT_CONFIRMATION}		
Description	Selects whether transmit confirmation notification is supported.		
Multiplicity	1		
Type	BooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: Module		

SWS Item	**--**		
Name	CanIfWakeupNotification {CANIF_DRIVER_WAKEUP_NOTIFICATION}		
Description	Selects whether wakeup indication notification is supported. True: Enabled False: Disabled		
Multiplicity	1		
Type	BooleanParamDef		
Default value	true		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfDriverNameRef {CANIF_DRIVER_VENDOR_ID}		
Description	CAN Interface Driver Reference. This reference can be used to get any information (Ex. Driver Name, Vendor ID) from the CAN driver. The CAN Driver name can be derived from the ShortName of the CAN driver module.		
Multiplicity	1		
Type	Reference to CanGeneral		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**--**		
Name	CanIfInitHohConfigRef		
Description	Reference to the Init Hoh Configuration		
Multiplicity	1		
Type	Reference to CanIfInitHohConfig		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency			

No Included Containers

10.2.12 CanIfTransceiverDrvConfig

SWS Item	**CANIF273 :**
Container Name	CanIfTransceiverDrvConfig{CanInterfaceTransceiverDriverConfiguration}
Description	This container contains the configuration (parameters) of all addressed CAN transceivers by each underlying CAN Transceiver Driver.
Configuration Parameters	

SWS Item	**--**
Name	CanIfTrcvWakeupNotification {CANIF_TRANSCEIVER_WAKEUP_NOTIFICATION}
Description	Selects whether wakeup indication notification is supported. True: Enabled False: Disabled
Multiplicity	1

Type	BooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**--**		
Name	CanIfTrcvIdRef		
Description	Logical handle of the underlying CAN transceiver to be served by the CAN Interface.		
Multiplicity	0..*		
Type	Reference to CanTrcvChannel		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU		

No Included Containers

10.2.13 CanIfInitHohConfig

SWS Item	**CANIF257 :**
Container Name	CanIfInitHohConfig
Description	This container contains the references to the configuration setup of each underlying CAN Driver.
Configuration Parameters	

SWS Item	**--**		
Name	CanIfRefConfigSet {CANIF_REF_CONFIGSET}		
Description	Selects the CAN Interface specific configuration setup. This type of external data structure shall contain the post build initialization data for the CAN Interface for all underlying CAN Drivers.		
Multiplicity	1		
Type	Reference to CanConfigSet		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Module		

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

10.2.14 CanIfHthConfig

SWS Item	**CANIF258 :**
Container Name	CanIfHthConfig{CanInterfaceHthConfiguration}
Description	This container contains parameters related to each HTH.
Configuration Parameters	

SWS Item	**--**		
Name	CanIfHthType {CANIF_HTH_TYPE}		
Description	Transmission method of the corresponding HTH.		
Multiplicity	1		
Type	EnumerationParamDef		
Range	BASIC_CAN	For a BasicCAN HTH buffers have to be reserved for buffering Tx L-PDUs. The size of Tx Queue is specified in parameter CanIfNumberOfTxBuffers.	
	FULL_CAN	The HTH is of type Full CAN. At the maximum only one Tx L-PDU should be assigned to this type of HTH.	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: This information has to be derived from the CAN Driver configuration		

SWS Item	**--**		
Name	CanIfCanControllerIdRef {CANIF_CONTROLLER_REF_ID}		
Description	Reference to controller Id to which the HTH belongs to. A controller can contain one or more HTHs.		
Multiplicity	1		
Type	Reference to CanIfControllerConfig		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency			

SWS Item	**--**		
Name	CanIfHthIdSymRef {CANIF_HTH_REF_ID}		
Description	The parameter refers to a particular HTH object in the CAN Driver Module configuration. The HTH id is unique in a given CAN Driver. The HTH Ids are defined in the CAN Driver Module and hence it is derived from CAN Driver Configuration.		
Multiplicity	1		
Type	Reference to CanHardwareObject		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency			

No Included Containers

10.2.15 CanIfHrhConfig

SWS Item	**CANIF259 :**
Container Name	CanIfHrhConfig{CanInterfaceHrhConfiguration}
Description	This container contains configuration parameters for each hardware receive object (HRH).
Configuration Parameters	

SWS Item	**--**		
Name	CanIfHrhType {CANIF_HRH_TYPE}		
Description	Defines the HRH type i.e, whether its a BasicCan or FullCan. If BasicCan is configured, software filtering is enabled.		
Multiplicity	1		
Type	EnumerationParamDef		
Range	BASIC_CAN	HRH is of type Basic CAN. More than one Rx L-PDUs can be assigned to same BasicCAN HRH.	
	FULL_CAN	HRH is of type Full CAN. At the maximum only one Rx L-PDU can be assigned to FullCAN type of HRH.	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency			

SWS Item	**--**		
Name	CanIfSoftwareFilterHrh {CANIF_SOFTWARE_FILTER_HRH}		
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	BooleanParamDef		
Default value	true		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency	scope: Module		

SWS Item	**--**		
Name	CanIfCanControllerHrhIdRef {CANIF_CONTROLLER_REF_ID}		
Description	Reference to controller Id to which the HRH belongs to. A controller can contain one or more HRHs.		
Multiplicity	1		
Type	Reference to CanIfControllerConfig		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME, VARIANT-POST-BUILD
	Post-build time	--	
Scope / Dependency			

SWS Item	**--**		
Name	CanIfHrhIdSymRef {CANIF_HRH_REF_ID}		
Description	The parameter refers to a particular HRH object in the CAN Driver Module configuration. The HRH id is unique in a given CAN Driver. The HRH Ids are defined in the CAN Driver Module and hence it is derived from CAN Driver Configuration.		
Multiplicity	1		
Type	Reference to CanHardwareObject		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME

	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency			

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanIfHrhRangeConfig	0..*	Defines the parameters required for configuraing multiple CANID ranges for a given same HRH.

10.2.16 CanIfHrhRangeConfig

SWS Item	**--**
Container Name	CanIfHrhRangeConfig{CanInterfaceHrhRangeConfiguration }
Description	Defines the parameters required for configuraing multiple CANID ranges for a given same HRH.
Configuration Parameters	

SWS Item	**--**		
Name	CanIfRxPduLowerCanId {CANIF_HRHRANGE_LOWER_CANID}		
Description	Lower CAN Identifier of a receive CAN L-PDU for identifier range definition, in which all CAN Ids shall pass the software filtering. Range: 11 Bit for Standard CAN Identifier 29 Bit for Extended CAN Identifer		
Multiplicity	1		
Type	IntegerParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Module		

SWS Item	**--**		
Name	CanIfRxPduUpperCanId {CANIF_HRHRANGE_UPPER_CANID}		
Description	Upper CAN Identifier of a receive CAN L-PDU for identifier range definition, in which all CAN Ids shall pass the software filtering. Range: 11 Bit for Standard CAN Identifier 29 Bit for Extended CAN Identifer		
Multiplicity	1		
Type	IntegerParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: Module		

No Included Containers

10.3 Published information

CANIF016: Published information contains data defined by the implementer of the SW module that does not change when the module is adapted (i.e. configured) to the actual HW/SW environment. It thus contains version and manufacturer information.

The following table lists configuration parameters that are published to be used in other BSW modules.

The standard common published information like

vendorId (<Module>_VENDOR_ID),
moduleId (<Module>_MODULE_ID),
arMajorVersion (<Module>_AR_MAJOR_VERSION),
arMinorVersion (<Module>_ AR_MINOR_VERSION),
arPatchVersion (<Module>_ AR_PATCH_VERSION),
swMajorVersion (<Module>_SW_MAJOR_VERSION),
swMinorVersion (<Module>_ SW_MINOR_VERSION),
swPatchVersion (<Module>_ SW_PATCH_VERSION),
vendorApiInfix (<Module>_VENDOR_API_INFIX)

is provided in the BSW Module Description Template (see [17] Figure 4.1 and Figure 7.1).

Additional published parameters are listed below if applicable for this module.

11 Changes to release 2.1

11.1 Deleted SWS items

SWS Item	*Rationale*
CANIF067	Network view now is provided by CAN State Manager
CANIF265	CanIf_WakeupSourceType deleted
CANIF269	Can Transceiver Driver API invoked by CanIf (Polling is now done by BSW Scheduler)
CANIF271-CANIF274	Network abstraction
CANIF039	CAN Driver requirement
CANIF061	Only requirement ID deleted; text is just a hint
CANIF241, CANIF242	No scheduled API anymore
CANIF128	No scheduled wakeup API anymore

11.2 Replaced SWS items

SWS Item of Release 2	*replaced by SWS Item*	*Rationale*

11.3 Changed SWS items

SWS Item	*Rationale*
CANIF026	Description improved for dynamic DLC usage
CANIF044	CANIF044 splitted into CANIF 044, CANIF291 and CANIF292
CANIF085	CAN Controller specific initialization is splitted up to CANIF293

11.4 Added SWS items

SWS Item	*Rationale*
CANIF283	Dependencies to EcuM described
CANIF284	Dependencies to BSW Scheduler described
CANIF285	Polling CAN device driver according refined wakeup detection and notification concept
CANIF286	Added for refined wakeup validation concept
CANIF287	Added for CanIf_SetTransceiverMode
CANIF288	Added for CanIf_GetTransceiverMode
CANIF289	Added for CanIf_GetTrcvWakeupReason
CANIF290	Added for CanIf_SetTransceiverWakeupMode
CANIF291	Splitted from CANIF044, HTH definition
CANIF292	Splitted from CANIF044, HRH definition
CANIF293	Splitted CANIF085, multiple CAN Controller in initialization

CANIF294	Optional interfaces
CANIF295	Added error classification
CANIF296	DLC check behavior with DLC=NULL
CANIF297	Number of bytes to be copied after DLC check
CANIF298	Change to STOPPED Mode in case of BusOff
CANIF299	Dependency to CanSM described.