Document Title	Specification of CAN Driver
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	011
Document Classification	Standard
Document Version	4.3.0
Document Status	Final
Part of Release	4.1
Revision	3

Document Change History			
Date	**Version**	**Changed by**	**Change Description**
31.03.2014	4.3.0	AUTOSAR Release Management	• Added new reqirements SWS_CAN_00497, SWS_CAN_00498, SWS_CAN_00499, and SWS_CAN_00496 • Modified reqirements ECUC_Can_00445, SWS_CAN_00487, SWS_CAN_00469, SWS_CAN_00475, and SWS_CAN_00479 • Removed reqirements SWS_CAN_00476, and SWS_Can_00414
31.10.2013	4.2.0	AUTOSAR Release Management	• Removed the 'Timing' row from the API table(s) of chapter 'Scheduled Functions' • Modified range of Can_IdType and CAN_CHANGE_BAUDRATE_SUPPORT to CAN_CHANGE_BAUDRATE_API • Editorial changes • Removed chapter(s) on change documentation

28.02.2013	4.1.0	AUTOSAR Administration	• Added support for Pretended Networking • Add DET error CAN_E_PARAM_BAUDRATE to the error classification table • Corrected the sequence for EcuM_SetWakeupEvent in section 7.7 • Updated Can_CheckWakeup as Configurable API • Added support to have more than one CanMailbox per HRH in order to receive back to back messages • Can_ChangeBaudrate and Can_CheckBaudrate API are deprecated and will be replaced by Can_SetBaudrate API
02.11.2011	4.0.0	AUTOSAR Administration	• Added SWS_Can_00461 to capture - Detection of Power ON of controller due to CAN communication • Changed Can_InitController to Can_ChangeBaudrate • Added Can_CheckBaudrate • Added sub container CanMainFunctionRWPeriods to CanGeneral • Changed CanHardwareObject container • Updated description of ECUC_Can_00321 • Changed Can_SetControllerMode in SWS_Can_00370 to Can_Mainfunction_Mode • Added CanControllerDefaultBaudrate parameter • Updated description of SWS_Can_00279 • Updated description of CAN321 • Added SWS_Can_00445, SWS_Can_00446 and SWS_Can_00447 to capture Possible loss of CAN Wakeup • Changed “Module Short Name” (MODULENAME) to “Module Abbreviation” (MAB)

15.10.2010	3.1.0	AUTOSAR Administration	• Modified SWS_Can_00111 to correct the “Version Checking” information • Added new requirements SWS_Can_00435 to SWS_Can_00440 to introduce Can_GeneralTypes.h. • Added new requirements SWS_Can_00441 and SWS_Can_00442 to introduce multiple poll cycles • Added new requirements SWS_Can_00443 and SWS_Can_00444 to provide an optional callback on every reception of a LPDU
30.11.2009	3.0.0	AUTOSAR Administration	• General improvements of requirements in preparation of CT-development. • Can_MainFunction_Mode added to support asynchronous controller state change • Limited number of supported message objects removed • Description of CAN controller state transitions improved • Debbuging concept added • Legal disclaimer revised
23.06.2008	2.2.2	AUTOSAR Administration	• Legal disclaimer revised
24.01.2008	2.2.1	AUTOSAR Administration	Table formatting corrected
30.11.2007	2.2.0	AUTOSAR Administration	- Tables generated from UML-models, - General improvements of requirements in preparation of CT-development. - Functions Can_MainFunction_Write, Can_MainFunction_Read, Can_MainFunction_BusOff and Can_MainFunction_WakeUp changed to scheduled functions - Cycle Parameters added for new scheduled functions - Wakeup concept added (Chapter 7.7) and addition of function Can_Cbk_CheckWakeup - Document meta information extended - Small layout adaptations made

31.01.2007	2.1.0	AUTOSAR Administration	- File structure reworked (chapter 5.5) - Removed return value CAN_WAKEUP in function Can_SetControllerMode - Replaced by CAN_NOT_OK - Renamed CanIf_ControllerWakeup to CanIf_SetWakeupEvent - Reworked development errors (chapter 0) - Removed implementation specific description in Can_Write - Changed timing of cyclic functions to “fixed cyclic” - Reworked “Scope” for all configuration variables (chapter 10.2) - Legal disclaimer revised - Release notes added - “Advice for users” revised - “Revision Information” added
21.04.2006	2.0.0	AUTOSAR Administration	Document structure adapted to common Release 2.0 SWS Template • clarified development and production error handling and function abortion • multiplexed transmission and TX cancellation • version check • configuration description according template • individual main functions for RX TX and status
31.05.2005	1.0.0	AUTOSAR Administration	Initial release

Disclaimer

Advice for users

AUTOSAR specifications 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 specifications for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such specifications, 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 Content

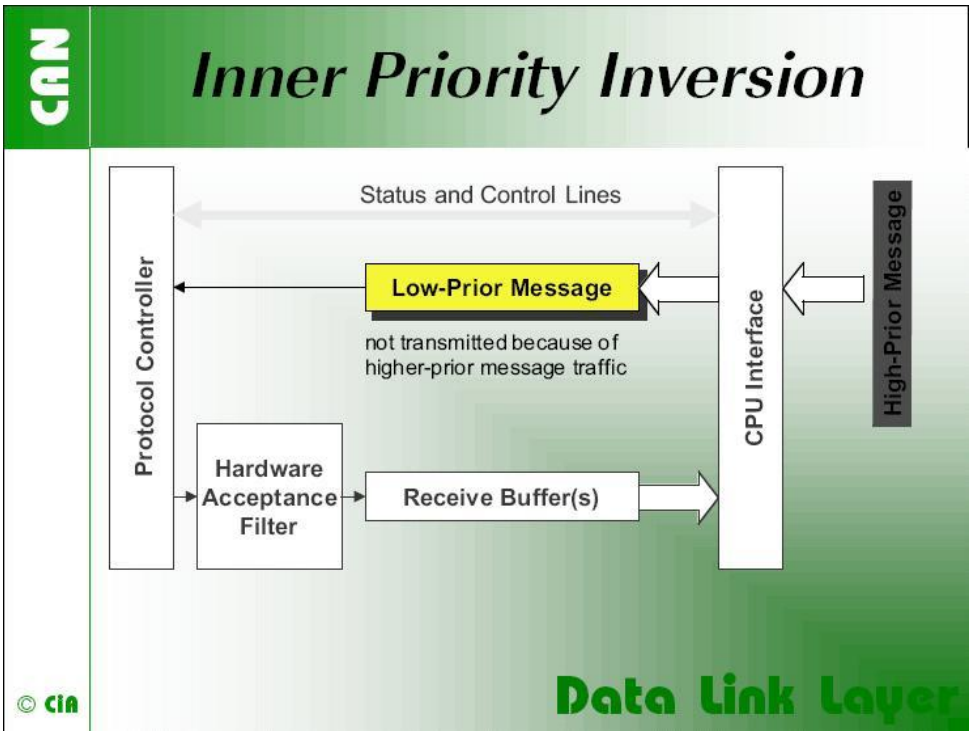
1 Introduction and functional overview

This specification specifies the functionality, API and the configuration of the AUTOSAR Basic Software module CAN Driver (called “Can module” in this document).
The Can module is part of the lowest layer, performs the hardware access and offers a hardware independent API to the upper layer.
The only upper layer that has access to the Can module is the CanIf module (see also SRS_SPAL_12092).
The Can module provides services for initiating transmissions and calls the callback functions of the CanIf module for notifying events, independently from the hardware.
Furthermore, it provides services to control the behavior and state of the CAN controllers that are belonging to the same CAN Hardware Unit.
Several CAN controllers can be controlled by a single Can module as long as they belong to the same CAN Hardware Unit.
For a closer description of CAN controller and CAN Hardware Unit see chapter Acronyms and abbreviations and a diagram in [5].

2 Acronyms and abbreviations

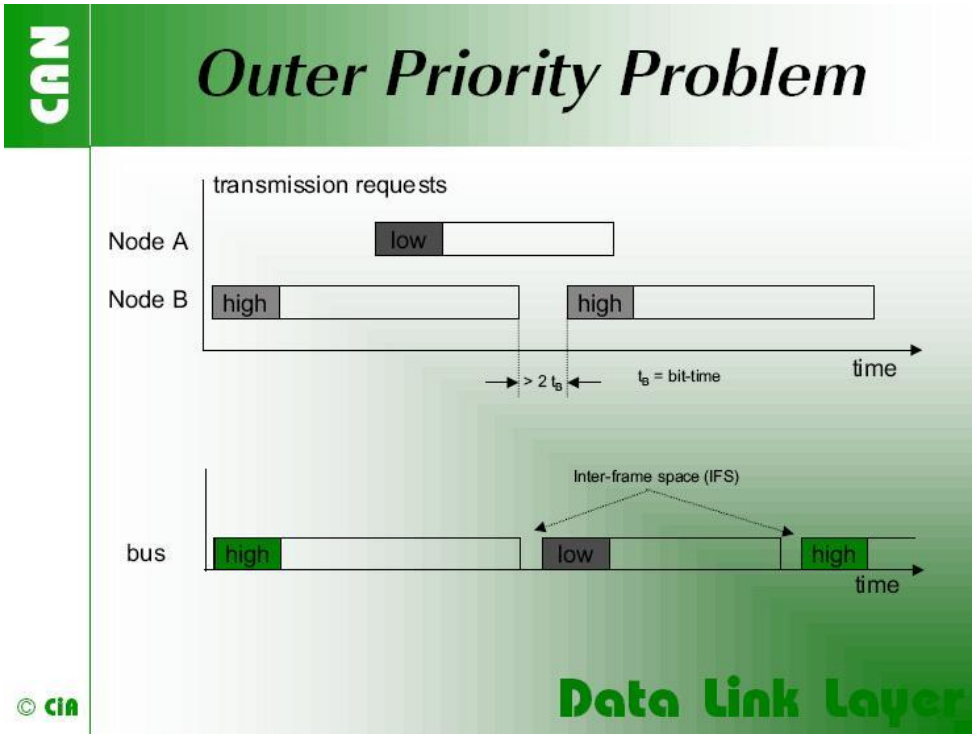
Abbreviation / Acronym:	*Description:*
CAN controller	A CAN controller serves exactly one physical channel.
CAN Hardware Unit	A CAN Hardware Unit may consists of one or multiple CAN controllers of the same type and one or multiple CAN RAM areas. The CAN Hardware Unit is either on-chip, or an external device. The CAN Hardware Unit is represented by one CAN driver.
CAN L-PDU	Data Link Layer Protocol Data Unit. Consists of Identifier, DLC and Data (SDU). (see [19])
CAN L-SDU	Data Link Layer Service Data Unit. Data that is transported inside the L-PDU. (see [19])
DLC	Data Length Code (part of L-PDU that describes the SDU length)
Hardware Object	A CAN hardware object is defined as a PDU buffer inside the CAN RAM of the CAN hardware unit / CAN controller. A Hardware Object is defined as L-PDU buffer inside the CAN RAM of the CAN Hardware Unit.
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 can be used to optimize 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.
ISR	Interrupt Service Routine
L-PDU Handle	The L-PDU handle is defined and placed inside the CanIf module layer. Typically each handle represents an L-PDU, which is a constant structure with information for Tx/Rx processing.
MCAL	Microcontroller Abstraction Layer
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.
Priority	The Priority of a CAN L-PDU is represented by the CAN Identifier. The lower the numerical value of the identifier, the higher the priority.
SFR	Special Function Register. Hardware register that controls the controller behavior.
SPAL	Standard Peripheral Abstraction Layer
ICOM	Intelligent Communication Controller

2.1 Priority Inversion



"If only a single transmit buffer is used inner priority inversion may occur. Because of low priority a message stored in the buffer waits until the "traffic on the bus calms down". During the waiting time this message could prevent a message of higher priority generated by the same microcontroller from being transmitted over the bus."[1]

[1] Picture and text by CiA (CAN in Automation)



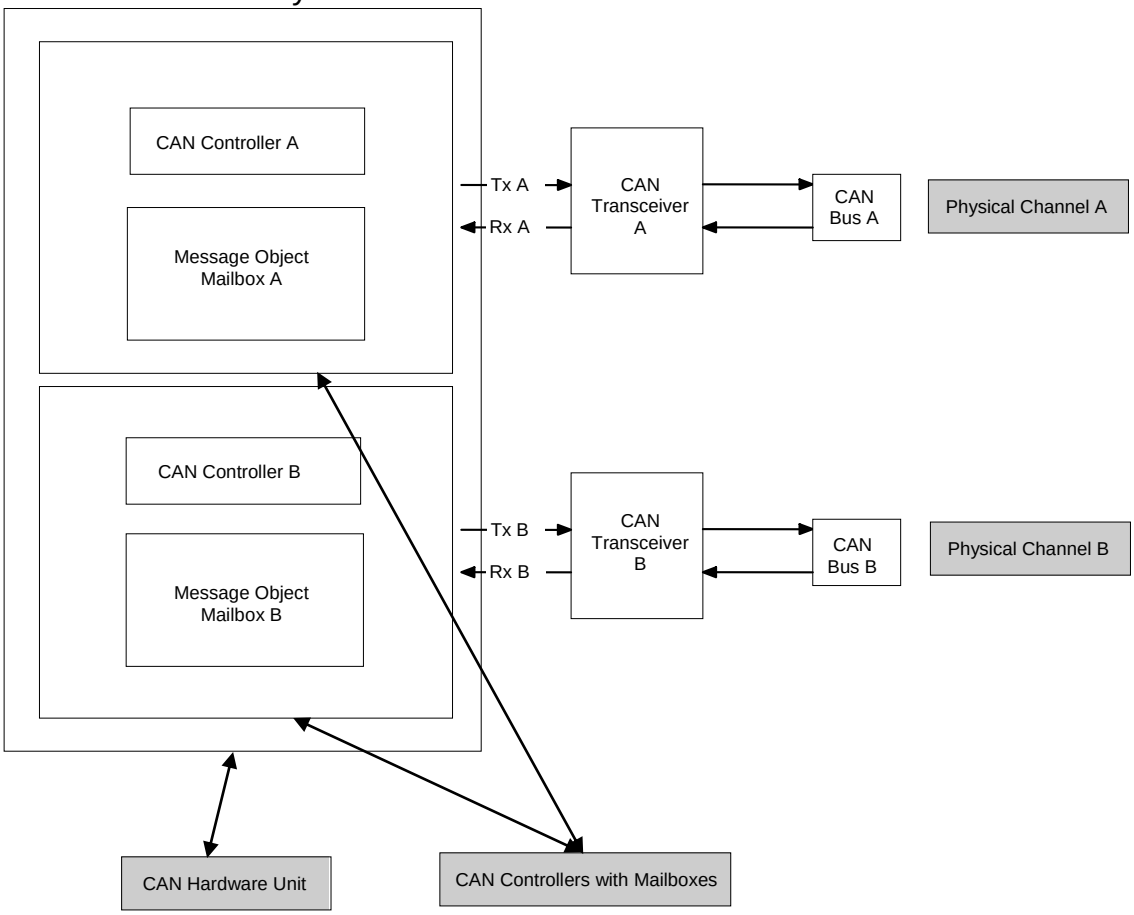
"The problem of outer priority inversion may occur in some CAN implementations. Let us assume that a CAN node wishes to transmit a package of consecutive messages with high priority, which are stored in different message buffers. If the interframe space between these messages on the CAN network is longer than the minimum space defined by the CAN standard, a second node is able to start the transmission of a lower priority message. The minimum interframe space is determined by the Intermission field, which consists of 3 recessive bits. A message, pending during the transmission of another message, is started during the Bus Idle period, at the earliest in the bit following the Intermission field. The exception is that a node with a waiting transmission message will interpret a dominant bit at the third bit of Intermission as Start-of-Frame bit and starts transmission with the first identifier bit without first transmitting an SOF bit. The internal processing time of a CAN module has to be short enough to send out consecutive messages with the minimum interframe space to avoid the outer priority inversion under all the scenarios mentioned."[2]

2.2 CAN Hardware Unit

The CAN Hardware Unit combines one or several CAN controllers, which may be located on-chip or as external standalone devices of the same type, with common or separate Hardware Objects.

[2] Text and image by CiA (CAN in Automation)

Following figure shows a CAN Hardware Unit consisting of two CAN controllers connected to two Physical Channels:



3 Related documentation

3.1 Input documents

[1] Layered Software Architecture
AUTOSAR_EXP_LayeredSoftwareArchitecture..pdf

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

[3] General Requirements on SPAL
AUTOSAR_SRS_SPALGeneral.pdf

[4] Requirements on CAN
AUTOSAR_SRS_CAN.pdf

[5] Specification of CAN Interface
AUTOSAR_SWS_CANInterface.pdf

[6] Specification of Development Error Tracer
AUTOSAR_SWS_DevelopmentErrorTracer.pdf

[7] Specification of ECU State Manager
AUTOSAR_SWS_ECUStateManager.pdf

[8] Specification of MCU Driver
AUTOSAR_SWS_MCUDriver.pdf

[9] Specification of Operating System
AUTOSAR_SWS_OS.pdf

[10] Specification of ECU Configuration
AUTOSAR_TPS_ECUConfiguration.pdf

[11] Specification of C Implementation Rules
AUTOSAR_TR_CImplementationRules.pdf

[12] Specification of SPI Handler/Driver
AUTOSAR_SWS_SPIHandlerDriver.doc.pdf

[13] Specification of Memory Mapping
AUTOSAR_SWS_MemoryMapping.pdf

[14] Specification of BSW Scheduler
AUTOSAR_SWS_BSW_Scheduler.pdf

[15] Basic Software Module Description Template
AUTOSAR_TPS_BSWModuleDescriptionTemplate.pdf

[16] List of Basis Software Modules
AUTOSAR_TR_BSWModuleList.pdf

[17] General Specification of Basic Software Modules
AUTOSAR_SWS_BSWGeneral.pdf

3.2 Related standards and norms

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

[19] ISO-IEC 7498-1 – OSI Basic Reference Model

[20] HIS – Joint Subset of the MISRA C Guidelines

3.3 Related specification

AUTOSAR provides a General Specification on Basic Software modules [17] (SWS BSW General), which is also valid for CAN Driver.

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

4 Constraints and assumptions

4.1 Limitations

A CAN controller always corresponds to one physical channel. It is allowed to connect physical channels on bus side. Regardless the CanIf module will treat the concerned CAN controllers separately.
A few CAN hardware units support the possibility to combine several CAN controllers by using the CAN RAM, to extend the number of message objects for one CAN controller. These combined CAN controller are handled as one controller by the Can module.
The Can module does not support CAN remote frames.

[SWS_Can_00237] 「The Can module shall not transmit messages triggered by remote transmission requests.」(SRS_Can_01147)

[SWS_Can_00236] 「The Can module shall initialize the CAN HW to ignore any remote transmission requests.」(SRS_Can_01147)

4.2 Applicability to car domains

The Can module can be used for any application, where the CAN protocol is used.

5 Dependencies to other modules

5.1 Static Configuration

The configuration elements described in chapter 10 can be referenced by other BSW modules for their configuration.

5.2 Driver Services

[SWS_Can_00238] ⌈If the CAN controller is on-chip, the Can module shall not use any service of other drivers.⌋(SRS_BSW_00005)

[SWS_Can_00239] ⌈The function Can_Init shall initialize all on-chip hardware resources that are used by the CAN controller. The only exception to this is the digital I/O pin configuration (of pins used by CAN), which is done by the port driver.⌋(SRS_BSW_00377)

[SWS_Can_00240] ⌈The Mcu module (SPAL see [8]) shall configure register settings that are 'shared' with other modules.⌋()

Implementation hint: The Mcu module shall be initialized before initializing the Can module.

[SWS_Can_00242] ⌈If an off-chip CAN controller is used[3], the Can module shall use services of other MCAL drivers (e.g. SPI).⌋(SRS_BSW_00005)

Implementation hint: If the Can module uses services of other MCAL drivers (e.g. SPI), it must be ensured that these drivers are up and running before initializing the Can module.
The sequence of initialization of different drivers is partly specified in [7].

[SWS_Can_00244] ⌈The Can module shall use the synchronous APIs of the underlying MCAL drivers and shall not provide callback functions that can be called by the MCAL drivers.⌋()

Thus the type of connection between µC and CAN Hardware Unit has only impact on implementation and not on the API.

5.3 System Services

[SWS_Can_00280] ⌈In special hardware cases, the Can module shall poll for events of the hardware.⌋()

[SWS_Can_00281] ⌈The Can module shall use the free running timer provided by the system service for timeout detection in case the hardware does not react in the expected time (hardware malfunction) to prevent endless loops.⌋()

[3] In this case the CAN driver is not any more part of the µC abstraction layer but put part of the ECU abstraction layer. Therefore it is (theoretically) allowed to use any µC abstraction layer driver it needs.

Implementation hint: The blocking time of the Can module function that is waiting for hardware reaction shall be shorter than the CAN main function (i.e. Can_MainFunction_Read) trigger period, because the CAN main functions can't be used for that purpose.

5.4 Can module Users

[SWS_Can_00058] 「The Can module interacts among other modules (eg. Diagnostic Event Manager (DEM), Development Error Tracer (DET), Ecu State Manager (ECUM)) with the CanIf module in a direct way. This document never specifies the actual origin of a request or the actual destination of a notification. The driver only sees the CanIf module as origin and destination.」(SRS_SPAL_12092)

5.5 File structure

5.5.1 Header file structure

[SWS_Can_00034] 「

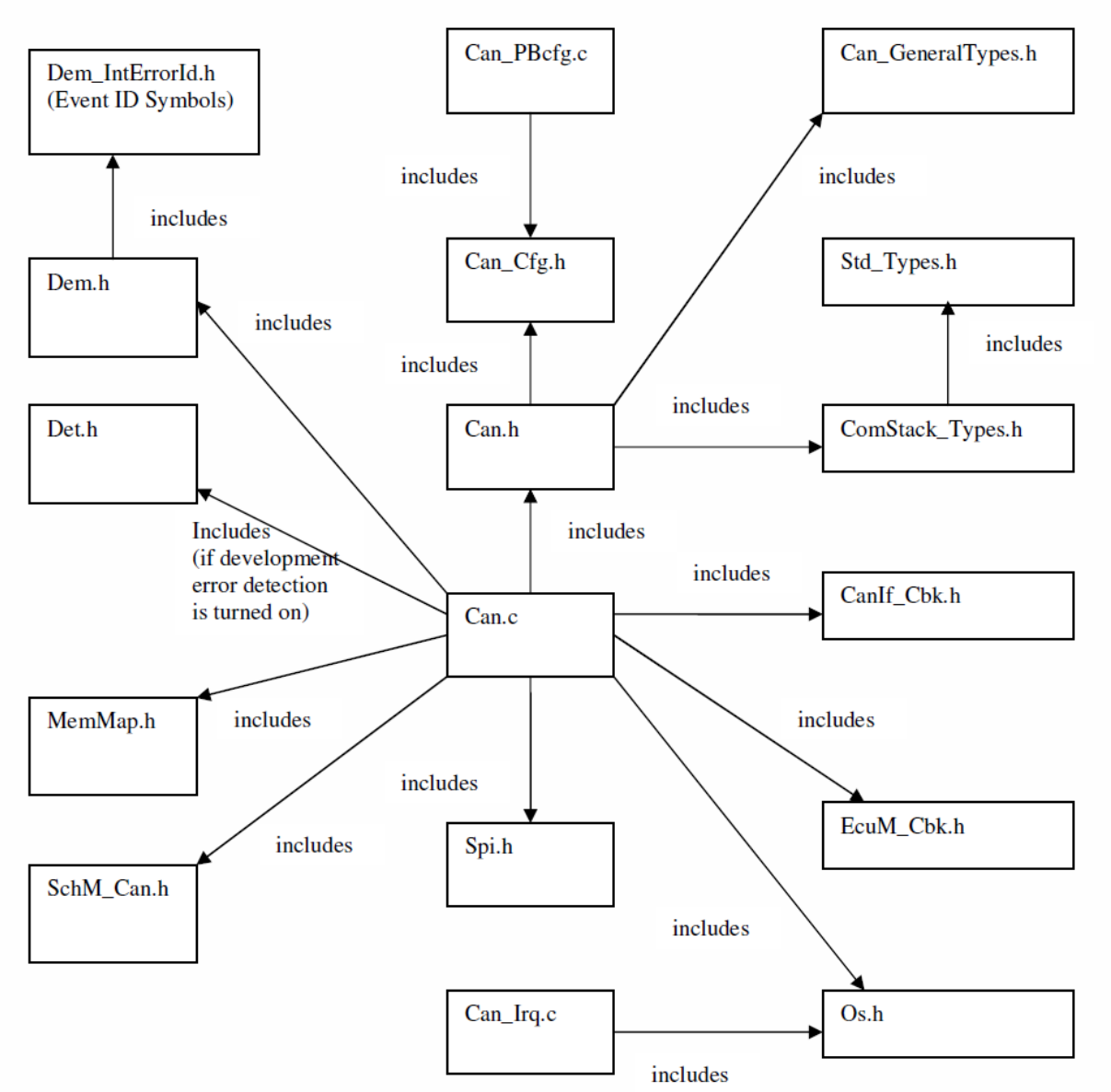


Figure 5-1: File structure for the Can module」(SRS_BSW_00381, SRS_BSW_00412, SRS_BSW_00346, SRS_BSW_00158, SRS_BSW_00435, SRS_BSW_00436, SRS_BSW_00348, SRS_BSW_00301)

[SWS_Can_00435]「The Can.h file shall include Can_GeneralTypes.h.」()

[SWS_Can_00436]「Can_GeneralTypes.h shall contain all types and constants that are shared among the AUTOSAR CAN modules Can, CanIf and CanTrcv.」()

[SWS_Can_00388]「The header file Can.h shall include the header file ComStack_Types.h.」()

[SWS_Can_00035] ⌈
The Can module does not provide callback functions (no Can_Cbk.h, see also SWS_Can_00244) ⌋()

[SWS_Can_00390] ⌈The Can module shall include the header file EcuM_Cbk.h, in which the callback functions called by the Can module at the Ecu State Manager module are declared.⌋()

[SWS_Can_00391] ⌈Can module implementations for off-chip CAN controllers shall include the header file Spi.h. By this inclusion, the APIs to access an external CAN controller by the SPI module [12] are included.⌋()

[SWS_Can_00397] ⌈The Can module shall include the header file Os.h file. By this inclusion, the API to read a free running timer value (GetCounterValue) provided by the system service shall be included.⌋()

6 Requirements traceability

Requirement	Description	Satisfied by
-	-	SWS_Can_00035
-	-	SWS_Can_00056
-	-	SWS_Can_00109
-	-	SWS_Can_00174
-	-	SWS_Can_00175
-	-	SWS_Can_00177
-	-	SWS_Can_00178
-	-	SWS_Can_00179
-	-	SWS_Can_00180
-	-	SWS_Can_00181
-	-	SWS_Can_00183
-	-	SWS_Can_00184
-	-	SWS_Can_00185
-	-	SWS_Can_00186
-	-	SWS_Can_00196
-	-	SWS_Can_00197
-	-	SWS_Can_00198
-	-	SWS_Can_00199
-	-	SWS_Can_00200
-	-	SWS_Can_00202
-	-	SWS_Can_00204
-	-	SWS_Can_00205
-	-	SWS_Can_00206
-	-	SWS_Can_00208
-	-	SWS_Can_00209
-	-	SWS_Can_00210
-	-	SWS_Can_00215
-	-	SWS_Can_00216
-	-	SWS_Can_00217
-	-	SWS_Can_00218
-	-	SWS_Can_00219
-	-	SWS_Can_00220
-	-	SWS_Can_00221
-	-	SWS_Can_00222
-	-	SWS_Can_00224
-	-	SWS_Can_00225

-	-	SWS_Can_00226
-	-	SWS_Can_00227
-	-	SWS_Can_00228
-	-	SWS_Can_00230
-	-	SWS_Can_00240
-	-	SWS_Can_00244
-	-	SWS_Can_00255
-	-	SWS_Can_00256
-	-	SWS_Can_00258
-	-	SWS_Can_00259
-	-	SWS_Can_00260
-	-	SWS_Can_00261
-	-	SWS_Can_00262
-	-	SWS_Can_00263
-	-	SWS_Can_00264
-	-	SWS_Can_00265
-	-	SWS_Can_00266
-	-	SWS_Can_00267
-	-	SWS_Can_00268
-	-	SWS_Can_00269
-	-	SWS_Can_00270
-	-	SWS_Can_00275
-	-	SWS_Can_00276
-	-	SWS_Can_00280
-	-	SWS_Can_00281
-	-	SWS_Can_00282
-	-	SWS_Can_00283
-	-	SWS_Can_00284
-	-	SWS_Can_00290
-	-	SWS_Can_00294
-	-	SWS_Can_00299
-	-	SWS_Can_00300
-	-	SWS_Can_00360
-	-	SWS_Can_00361
-	-	SWS_Can_00362
-	-	SWS_Can_00363
-	-	SWS_Can_00368
-	-	SWS_Can_00369
-	-	SWS_Can_00370

-	-	SWS_Can_00373
-	-	SWS_Can_00379
-	-	SWS_Can_00384
-	-	SWS_Can_00385
-	-	SWS_Can_00386
-	-	SWS_Can_00388
-	-	SWS_Can_00390
-	-	SWS_Can_00391
-	-	SWS_Can_00395
-	-	SWS_Can_00397
-	-	SWS_Can_00398
-	-	SWS_Can_00404
-	-	SWS_Can_00405
-	-	SWS_Can_00408
-	-	SWS_Can_00409
-	-	SWS_Can_00410
-	-	SWS_Can_00411
-	-	SWS_Can_00412
-	-	SWS_Can_00413
-	-	SWS_Can_00415
-	-	SWS_Can_00416
-	-	SWS_Can_00417
-	-	SWS_Can_00419
-	-	SWS_Can_00420
-	-	SWS_Can_00422
-	-	SWS_Can_00423
-	-	SWS_Can_00425
-	-	SWS_Can_00426
-	-	SWS_Can_00427
-	-	SWS_Can_00429
-	-	SWS_Can_00432
-	-	SWS_Can_00433
-	-	SWS_Can_00434
-	-	SWS_Can_00435
-	-	SWS_Can_00436
-	-	SWS_Can_00439
-	-	SWS_Can_00440
-	-	SWS_Can_00441
-	-	SWS_Can_00442

-	-	SWS_Can_00443
-	-	SWS_Can_00444
-	-	SWS_Can_00445
-	-	SWS_Can_00446
-	-	SWS_Can_00447
-	-	SWS_Can_00449
-	-	SWS_Can_00450
-	-	SWS_Can_00451
-	-	SWS_Can_00452
-	-	SWS_Can_00453
-	-	SWS_Can_00454
-	-	SWS_Can_00455
-	-	SWS_Can_00456
-	-	SWS_Can_00457
-	-	SWS_Can_00458
-	-	SWS_Can_00459
-	-	SWS_Can_00460
-	-	SWS_Can_00461
-	-	SWS_CAN_00462
-	-	SWS_CAN_00463
-	-	SWS_CAN_00464
-	-	SWS_CAN_00465
-	-	SWS_CAN_00466
-	-	SWS_CAN_00467
-	-	SWS_CAN_00468
-	-	SWS_CAN_00469
-	-	SWS_CAN_00470
-	-	SWS_CAN_00471
-	-	SWS_CAN_00472
-	-	SWS_CAN_00473
-	-	SWS_CAN_00474
-	-	SWS_CAN_00475
-	-	SWS_CAN_00477
-	-	SWS_CAN_00478
-	-	SWS_CAN_00479
-	-	SWS_CAN_00480
-	-	SWS_CAN_00481
-	-	SWS_CAN_00482
-	-	SWS_CAN_00483

-	-	SWS_CAN_00484
-	-	SWS_CAN_00485
-	-	SWS_CAN_00486
-	-	SWS_CAN_00487
-	-	SWS_CAN_00489
-	-	SWS_CAN_00490
-	-	SWS_CAN_00491
-	-	SWS_CAN_00492
-	-	SWS_CAN_00493
-	-	SWS_CAN_00494
-	-	SWS_CAN_00495
-	-	SWS_CAN_00496
-	-	SWS_CAN_00497
-	-	SWS_CAN_00498
-	-	SWS_CAN_00499
BSW00443	-	SWS_Can_00999
BSW00444	-	SWS_Can_00999
BSW00445	-	SWS_Can_00999
BSW00446	-	SWS_Can_00999
SRS_BSW_00005	Modules of the æC Abstraction Layer (MCAL) may not have hard coded horizontal interfaces	SWS_Can_00238, SWS_Can_00242
SRS_BSW_00007	All Basic SW Modules written in C language shall conform to the MISRA C 2004 Standard.	SWS_Can_00079
SRS_BSW_00101	The Basic Software Module shall be able to initialize variables and hardware in a separate initialization function	SWS_Can_00250
SRS_BSW_00158	All modules of the AUTOSAR Basic Software shall strictly separate configuration from implementation	SWS_Can_00034
SRS_BSW_00159	All modules of the AUTOSAR Basic Software shall support a tool based configuration	SWS_Can_00022
SRS_BSW_00162	The AUTOSAR Basic Software shall provide a hardware abstraction layer	SWS_Can_00999
SRS_BSW_00164	The Implementation of interrupt service routines shall be done by the Operating System, complex drivers or modules	SWS_Can_00033
SRS_BSW_00167	All AUTOSAR Basic Software Modules shall provide configuration rules and constraints to enable plausibility checks	SWS_Can_00024
SRS_BSW_00168	SW components shall be tested by a function defined in a common API in the Basis-SW	SWS_Can_00999

SRS_BSW_00170	The AUTOSAR SW Components shall provide information about their dependency from faults, signal qualities, driver demands	SWS_Can_00999
SRS_BSW_00301	All AUTOSAR Basic Software Modules shall only import the necessary information	SWS_Can_00034
SRS_BSW_00306	AUTOSAR Basic Software Modules shall be compiler and platform independent	SWS_Can_00079
SRS_BSW_00307	Global variables naming convention	SWS_Can_00999
SRS_BSW_00308	AUTOSAR Basic Software Modules shall not define global data in their header files, but in the C file	SWS_Can_00079
SRS_BSW_00309	All AUTOSAR Basic Software Modules shall indicate all global data with read-only purposes by explicitly assigning the const keyword	SWS_Can_00079
SRS_BSW_00312	Shared code shall be reentrant	SWS_Can_00214, SWS_Can_00231, SWS_Can_00232, SWS_Can_00233
SRS_BSW_00323	All AUTOSAR Basic Software Modules shall check passed API parameters for validity	SWS_Can_00026
SRS_BSW_00325	The runtime of interrupt service routines and functions that are running in interrupt context shall be kept short	SWS_Can_00999
SRS_BSW_00326	-	SWS_Can_00999
SRS_BSW_00330	It shall be allowed to use macros instead of functions where source code is used and runtime is critical	SWS_Can_00079
SRS_BSW_00331	All Basic Software Modules shall strictly separate error and status information	SWS_Can_00039, SWS_Can_00104
SRS_BSW_00336	Basic SW module shall be able to shutdown	SWS_Can_00999
SRS_BSW_00337	Classification of development errors	SWS_Can_00026, SWS_Can_00104
SRS_BSW_00342	It shall be possible to create an AUTOSAR ECU out of modules provided as source code and modules provided as object code, even mixed	SWS_Can_00999
SRS_BSW_00344	BSW Modules shall support link-time configuration	SWS_Can_00021
SRS_BSW_00346	All AUTOSAR Basic Software Modules shall provide at least a basic set of module files	SWS_Can_00034
SRS_BSW_00347	A Naming seperation of different instances of BSW drivers shall be in place	SWS_Can_00077
SRS_BSW_00348	All AUTOSAR standard types and constants shall be placed and organized in a standard type header file	SWS_Can_00034

SRS_BSW_00353	All integer type definitions of target and compiler specific scope shall be placed and organized in a single type header	SWS_Can_00999
SRS_BSW_00358	The return type of init() functions implemented by AUTOSAR Basic Software Modules shall be void	SWS_Can_00223
SRS_BSW_00359	All AUTOSAR Basic Software Modules callback functions shall avoid return types other than void if possible	SWS_Can_00999
SRS_BSW_00361	All mappings of not standardized keywords of compiler specific scope shall be placed and organized in a compiler specific type and keyword header	SWS_Can_00999
SRS_BSW_00369	All AUTOSAR Basic Software Modules shall not return specific development error codes via the API	SWS_Can_00089
SRS_BSW_00373	The main processing function of each AUTOSAR Basic Software Module shall be named according the defined convention	SWS_Can_00031
SRS_BSW_00375	Basic Software Modules shall report wake-up reasons	SWS_Can_00271, SWS_Can_00364
SRS_BSW_00376	-	SWS_Can_00031
SRS_BSW_00377	A Basic Software Module can return a module specific types	SWS_Can_00239
SRS_BSW_00378	AUTOSAR shall provide a boolean type	SWS_Can_00999
SRS_BSW_00381	The pre-compile time parameters shall be placed into a separate configuration header file	SWS_Can_00034
SRS_BSW_00383	The Basic Software Module specifications shall specify which other configuration files from other modules they use at least in the description	SWS_Can_00999
SRS_BSW_00385	List possible error notifications	SWS_Can_00104
SRS_BSW_00386	The BSW shall specify the configuration for detecting an error	SWS_Can_00089
SRS_BSW_00387	The Basic Software Module specifications shall specify how the callback function is to be implemented	SWS_Can_00234
SRS_BSW_00395	The Basic Software Module specifications shall list all configuration parameter dependencies	SWS_Can_00999
SRS_BSW_00397	The configuration parameters in pre-compile time are fixed before compilation starts	SWS_Can_00999
SRS_BSW_00398	The link-time configuration is achieved on object code basis in the stage after compiling and before linking	SWS_Can_00999
SRS_BSW_00399	Parameter-sets shall be located in a	SWS_Can_00999

	separate segment and shall be loaded after the code	
SRS_BSW_00400	Parameter shall be selected from multiple sets of parameters after code has been loaded and started	SWS_Can_00999
SRS_BSW_00404	BSW Modules shall support post-build configuration	SWS_Can_00021
SRS_BSW_00405	BSW Modules shall support multiple configuration sets	SWS_Can_00021
SRS_BSW_00406	A static status variable denoting if a BSW module is initialized shall be initialized with value 0 before any APIs of the BSW module is called	SWS_Can_00103
SRS_BSW_00409	All production code error ID symbols are defined by the Dem module and shall be retrieved by the other BSW modules from Dem configuration	SWS_Can_00999
SRS_BSW_00412	References to c-configuration parameters shall be placed into a separate h-file	SWS_Can_00034
SRS_BSW_00413	An index-based accessing of the instances of BSW modules shall be done	SWS_Can_00999
SRS_BSW_00414	The init function may have parameters	SWS_Can_00223
SRS_BSW_00415	Interfaces which are provided exclusively for one module shall be separated into a dedicated header file	SWS_Can_00999
SRS_BSW_00417	Software which is not part of the SW-C shall report error events only after the DEM is fully operational.	SWS_Can_00999
SRS_BSW_00422	Pre-de-bouncing of error status information is done within the DEM	SWS_Can_00999
SRS_BSW_00423	BSW modules with AUTOSAR interfaces shall be describable with the means of the SW-C Template	SWS_Can_00999
SRS_BSW_00424	BSW module main processing functions shall not be allowed to enter a wait state	SWS_Can_00999
SRS_BSW_00425	The BSW module description template shall provide means to model the defined trigger conditions of schedulable objects	SWS_Can_00999
SRS_BSW_00426	BSW Modules shall ensure data consistency of data which is shared between BSW modules	SWS_Can_00999
SRS_BSW_00427	ISR functions shall be defined and documented in the BSW module description template	SWS_Can_00999
SRS_BSW_00428	A BSW module shall state if its main processing function(s) has to be executed in a specific order or sequence	SWS_Can_00110

SRS_BSW_00429	BSW modules shall be only allowed to use OS objects and/or related OS services	SWS_Can_00999
SRS_BSW_00432	Modules should have separate main processing functions for read/receive and write/transmit data path	SWS_Can_00031, SWS_Can_00108, SWS_Can_00112
SRS_BSW_00433	Main processing functions are only allowed to be called from task bodies provided by the BSW Scheduler	SWS_Can_00999
SRS_BSW_00435	-	SWS_Can_00034
SRS_BSW_00436	-	SWS_Can_00034
SRS_BSW_00438	Configuration data shall be defined in a structure	SWS_Can_00291
SRS_BSW_00439	Enable BSW modules to handle interrupts	SWS_Can_00999
SRS_BSW_00440	The callback function invocation by the BSW module shall follow the signature provided by RTE to invoke servers via Rte_Call API	SWS_Can_00999
SRS_BSW_00447	Standardizing Include file structure of BSW Modules Implementing Autosar Service	SWS_Can_00999
SRS_BSW_00449	BSW Service APIs used by Autosar Application Software shall return a Std_ReturnType	SWS_Can_00999
SRS_BSW_00453	BSW Modules shall be harmonized	SWS_Can_00999
SRS_BSW_00455	-	SWS_Can_00999
SRS_Can_01041	The CAN Driver shall implement an interface for initialization	SWS_Can_00245, SWS_Can_00246
SRS_Can_01042	The CAN Driver shall support dynamic selection of configuration sets	SWS_Can_00062
SRS_Can_01043	The CAN Driver shall provide a service to enable/disable interrupts of the CAN Controller.	SWS_Can_00049, SWS_Can_00050
SRS_Can_01045	The CAN Driver shall offer a reception indication service.	SWS_Can_00279, SWS_Can_00396
SRS_Can_01049	The CAN Driver shall provide a dynamic transmission request service	SWS_Can_00212, SWS_Can_00213, SWS_Can_00214
SRS_Can_01051	The CAN Driver shall provide a transmission confirmation service	SWS_Can_00016
SRS_Can_01053	The CAN Driver shall provide a service to change the CAN controller mode.	SWS_Can_00017
SRS_Can_01054	The CAN Driver shall provide a notification for controller wake-up events	SWS_Can_00235, SWS_Can_00271, SWS_Can_00364
SRS_Can_01055	The CAN Driver shall provide a notification for bus-off state	SWS_Can_00020, SWS_Can_00234
SRS_Can_01059	The CAN Driver shall guarantee data consistency of received L-PDUs	SWS_Can_00011, SWS_Can_00012

SRS_Can_01060	The CAN driver shall not recover from bus-off automatically	SWS_Can_00272, SWS_Can_00273, SWS_Can_00274
SRS_Can_01062	Each event for each CAN Controller shall be configurable to be detected by polling or by an interrupt	SWS_Can_00007
SRS_Can_01122	The CAN driver shall support the situation where a wakeup by bus occurs during the same time the transition to standby/sleep is in progress	SWS_Can_00048
SRS_Can_01125	The CAN stack shall ensure not to lose messages in receive direction	SWS_Can_00999
SRS_Can_01126	The CAN stack shall be able to produce 100% bus load	SWS_Can_00999
SRS_Can_01132	The CAN driver shall be able to detect notification events message object specific by CAN-Interrupt and polling	SWS_Can_00099
SRS_Can_01133	The CAN driver shall support the HW Transmit Cancellation	SWS_Can_00278, SWS_Can_00285, SWS_Can_00286, SWS_Can_00287, SWS_Can_00288, SWS_Can_00399, SWS_Can_00400
SRS_Can_01134	The CAN Driver shall support multiplexed transmission	SWS_Can_00277, SWS_Can_00401, SWS_Can_00402, SWS_Can_00403
SRS_Can_01135	It shall be possible to configure one or several TX Hardware Objects	SWS_Can_00100
SRS_Can_01139	The CAN Interface and Driver shall offer a CAN Controller specific interface for initialization	SWS_Can_00062
SRS_Can_01147	The CAN Driver shall not support remote frames	SWS_Can_00236, SWS_Can_00237
SRS_SPAL_00157	All drivers and handlers of the AUTOSAR Basic Software shall implement notification mechanisms of drivers and handlers	SWS_Can_00026, SWS_Can_00031, SWS_Can_00108, SWS_Can_00112
SRS_SPAL_12056	All driver modules shall allow the static configuration of notification mechanism	SWS_Can_00235
SRS_SPAL_12057	All driver modules shall implement an interface for initialization	SWS_Can_00245, SWS_Can_00246
SRS_SPAL_12063	All driver modules shall only support raw value mode	SWS_Can_00059, SWS_Can_00060
SRS_SPAL_12064	All driver modules shall raise an error if the change of the operation mode leads to degradation of running operations	SWS_Can_00999
SRS_SPAL_12067	All driver modules shall set their wake-up conditions depending on the selected operation mode	SWS_Can_00257
SRS_SPAL_12068	The modules of the MCAL shall be initialized in a defined sequence	SWS_Can_00999
SRS_SPAL_12069	All drivers of the SPAL that wake up from a wake-up interrupt shall report the wake-up reason	SWS_Can_00271, SWS_Can_00364

SRS_SPAL_12075	All drivers with random streaming capabilities shall use application buffers	SWS_Can_00011
SRS_SPAL_12077	All drivers shall provide a non blocking implementation	SWS_Can_00371, SWS_Can_00372
SRS_SPAL_12092	The driver's API shall be accessed by its handler or manager	SWS_Can_00058
SRS_SPAL_12125	All driver modules shall only initialize the configured resources	SWS_Can_00053
SRS_SPAL_12129	The ISRs shall be responsible for resetting the interrupt flags and calling the according notification function	SWS_Can_00033
SRS_SPAL_12163	All driver modules shall implement an interface for de-initialization	SWS_Can_00999
SRS_SPAL_12169	All driver modules that provide different operation modes shall provide a service for mode selection	SWS_Can_00017
SRS_SPAL_12263	The implementation of all driver modules shall allow the configuration of specific module parameter types at link time	SWS_Can_00021
SRS_SPAL_12265	Configuration data shall be kept constant	SWS_Can_00021
SRS_SPAL_12448	All driver modules shall have a specific behavior after a development error detection	SWS_Can_00089, SWS_Can_00091
SRS_SPAL_12461	Specific rules regarding initialization of controller registers shall apply to all driver implementations	SWS_Can_00407
SRS_SPAL_12462	The register initialization settings shall be published	SWS_Can_00999
SRS_SPAL_12463	The register initialization settings shall be combined and forwarded	SWS_Can_00024

Document: General requirements on Basic Software [2]

Requirement	***Satisfied by***
[SRS_BSW_00344] Reference to link-time configuration	SWS_Can_00021
[SRS_BSW_00404] Reference to post build time configuration	SWS_Can_00021
[SRS_BSW_00405] Reference to multiple configuration sets	SWS_Can_00021
[SRS_BSW_00345] Pre-Build Configuration	SWS_Can_00389
[SRS_BSW_00159] Tool-based configuration	SWS_Can_00022
[SRS_BSW_00167] Static configuration checking	SWS_Can_00023, SWS_Can_00024
[SRS_BSW_00171] Configurability of optional functionality	ECUC_Can_00064, ECUC_Can_00095, ECUC_Can_00069
[SRS_BSW_00170] Data for reconfiguration of SW-components	not applicable (doesn't concern this document)
[SRS_BSW_00380] C-Files for configuration parameters	SWS_Can_00078
[SRS_BSW_00419] Separate C-Files for pre-compile time configuration	SWS_Can_00078

[SRS_BSW_00381] Separate configuration header file for pre-compile time parameters	SWS_Can_00034
[SRS_BSW_00412] Separate H-File for configuration parameters	SWS_Can_00034
[SRS_BSW_00383] List dependencies of configuration files	not applicable (implementation specific documentation)
[SRS_BSW_00384] List dependencies to other modules	Chapter 5
[SRS_BSW_00387] Specify the configuration class of callback function	SWS_Can_00234
[SRS_BSW_00388] Introduce containers	Chapter 10.2
[SRS_BSW_00389] Containers shall have names	Chapter 10.2
[SRS_BSW_00390] Parameter content shall be unique within the module	Chapter 10.2
[SRS_BSW_00391] Parameter shall have unique names	Chapter 10.2
[SRS_BSW_00392] Parameters shall have a type	Chapter 10.2
[SRS_BSW_00393] Parameters shall have a range	Chapter 10.2
[SRS_BSW_00394] Specify the scope of the parameters	Chapter 10.2
[SRS_BSW_00395] List the required parameters	not applicable (the parameters are defined in a way that their values are independent from other settings. The dependency is in the code generation (implementation) not in the configuration description -> hardware abstraction)
[SRS_BSW_00396] Configuration classes	Chapter 10.2
[SRS_BSW_00397] Pre-compile-time parameters	Not applicable: this is not a requirement but a definition of term.
[SRS_BSW_00398] Link-time parameters	Not applicable: this is not a requirement but a definition of term.
[SRS_BSW_00399] Loadable Post-build time parameters	Not applicable: this is not a requirement but a definition of term.
[SRS_BSW_00400] Selectable Post-build time parameters	Not applicable: this is not a requirement but a definition of term.
[SRS_BSW_00438] Post Build Configuration Data Structure	SWS_Can_00291
[SRS_BSW_00402] Published information	Chapter 10.2.2
[SRS_BSW_00375] Notification of wake-up reason	SWS_Can_00271, SWS_Can_00364
[SRS_BSW_00101] Initialization interface	SWS_Can_00250
[SRS_BSW_00168] Diagnostic Interface of SW components	not applicable (requirement for the diagnostic services, not for the BSW module)
[SRS_BSW_00416] Sequence of Initialization	not applicable (this is a general software integration requirement)
[SRS_BSW_00406] Check module initialization	SWS_Can_00103, defined development error CAN_E_UNINIT
[SRS_BSW_00437] NoInit—Area in RAM	not applicable
[SRS_BSW_00407] Function to read out published parameters	SWS_Can_00105, ECUC_Can_00106
[SRS_BSW_00423] Usage of SW-C template to describe BSW modules with AUTOSAR Interfaces	not applicable (this module does not provide an AUTOSAR interface)
[SRS_BSW_00424] BSW main processing function task allocation	not applicable (requirement on system design, not on a single module)

[SRS_BSW_00425] Trigger conditions for schedulable objects	not applicable (trigger conditions are system configuration specific.)
[SRS_BSW_00426] Exclusive areas in BSW modules	not applicable (no exclusive areas defined)
[SRS_BSW_00427] ISR description for BSW modules	not applicable (no ISR's defined for this module, usage of interrupts is implementation specific)
[SRS_BSW_00428] Execution order dependencies of main processing functions	SWS_Can_00110
[SRS_BSW_00429] Restricted BSW OS functionality access	not applicable (requirement on the implementation, not for the specification)
[SRS_BSW_00432] Modules should have separate main processing functions for read/receive and write/transmit data path	SWS_Can_00031, SWS_Can_00108, SWS_Can_00109, SWS_Can_00112
[SRS_BSW_00433] Calling of main processing functions	not applicable (requirement on system design, not on a single module)
[SRS_BSW_00450] Main Function Processing for Un-Initialized Modules	SWS_Can_00431
[SRS_BSW_00442] Debugging Support in Modules	SWS_Can_00365, SWS_Can_00366, SWS_Can_00367
[SRS_BSW_00336] Shutdown interface	not applicable
[SRS_BSW_00337] Classification of errors	SWS_Can_00026, SWS_Can_00027, SWS_Can_00028, SWS_Can_00104
[SRS_BSW_00338] Detection and Reporting of development errors	SWS_Can_00028, SWS_Can_00027
[SRS_BSW_00369] Do not return development error codes via API	SWS_Can_00089
[SRS_BSW_00339] Reporting of production relevant errors and exceptions	ECUC_Can_00113
[SRS_BSW_00422] Debouncing of production relevant error status	not applicable (requirement on the DEM)
[SRS_BSW_00417] Reporting of Error Events by Non-Basic Software	not applicable (this is a BSW module)
[SRS_BSW_00323] API parameter checking	SWS_Can_00026
[SRS_BSW_00004] Version check	SWS_Can_00111
[SRS_BSW_00409] Header files for production code error IDs	not applicable (no production errors codes used by Can module)
[SRS_BSW_00385] List possible error notifications	SWS_Can_00104
[SRS_BSW_00386] Configuration for detecting an error	SWS_Can_00089
[SRS_BSW_00455] Implementation Conformance Class 1 and 2 (ICC1 and ICC2) Guidelines	not applicable
[SRS_BSW_00161] Microcontroller abstraction	Chapter 1
[SRS_BSW_00162] ECU layout abstraction	not applicable (done in CanIf module)
[SRS_BSW_00005] No hard coded horizontal interfaces within MCAL	SWS_Can_00238, SWS_Can_00242
[SRS_BSW_00415] User dependent include files	not applicable (only one user for this module)
[SRS_BSW_00164] Implementation of interrupt service routines	SWS_Can_00033
[SRS_BSW_00325] Runtime of interrupt service routines	not applicable (The runtime is not under control of the Can module, because callback functions are called.)

[SRS_BSW_00326] Transition from ISRs to OS tasks	not applicable. When the transition from ISR to OS task is done will be defined in COM Stack SWS
[SRS_BSW_00342] Usage of source code and object code	not applicable (Only source code delivery is supported)
[SRS_BSW_00343] Specification and configuration of time	ECUC_Can_00113, ECUC_Can_00355, ECUC_Can_00356, ECUC_Can_00357, ECUC_Can_00358, ECUC_Can_00376
[SRS_BSW_00160] Human-readable configuration data	SWS_Can_00047
[SRS_BSW_00453] Harmonization of BSW Modules	not applicable, yet
[SRS_BSW_00007] HIS MISRA C	SWS_Can_00079
[SRS_BSW_00300] Module naming convention	is fulfilled, see function definitions in 0
[SRS_BSW_00413] Accessing instances of BSW modules	not applicable (this requirement is fulfilled by the CanIf module specification)
[SRS_BSW_00347] Naming separation of drivers	SWS_Can_00077
[SRS_BSW_00441] Enumeration literals and #define naming convention	Chapter 8.2.6, Chapter 8.2.7
[SRS_BSW_00305] Self-defined data types naming convention	is fulfilled, see type definitions in 8.2
[SRS_BSW_00307] Global variables naming convention	not applicable (because no global variables are specified for Can module)
[SRS_BSW_00310] API naming convention	is fulfilled, see function definitions in 0
[SRS_BSW_00373] Main processing function naming convention	SWS_Can_00031
[SRS_BSW_00327] Error values naming convention	Chapter 7.11.1 error names have been selected accordingly
[SRS_BSW_00335] Status values naming convention	Chapter 7.2 is fulfilled by state description
[SRS_BSW_00350] Development error detection keyword	ECUC_Can_00064
[SRS_BSW_00408] Configuration parameter naming convention	Chapter 10.2
[SRS_BSW_00410] Compiler switches shall have defined values	Chapter 10.2
[SRS_BSW_00411] Get version info keyword	ECUC_Can_00106
[SRS_BSW_00346] Basic set of module files	SWS_Can_00034
[SRS_BSW_00158] Separation of configuration from implementation	SWS_Can_00034
[SRS_BSW_00314] Separation of interrupt frames and service routines	SWS_Can_00035
[SRS_BSW_00370] Separation of callback interface from API	SWS_Can_00036
[SRS_BSW_00435] Module Header File Structure for the Basic Software Scheduler	SWS_Can_00034, SWS_Can_00406
[SRS_BSW_00436] Module Header File Structure for the Basic Software Memory Mapping	SWS_Can_00034, SWS_Can_00394
[SRS_BSW_00447] Standardizing Include file structure of BSW Modules Implementing Autosar Service	not applicable
[SRS_BSW_00348] Standard type header	SWS_Can_00034
[SRS_BSW_00353] Platform specific type header	not applicable (automatically included with Standard types)
[SRS_BSW_00361] Compiler specific language extension header	not applicable
[SRS_BSW_00301] Limit imported information	SWS_Can_00034

[SRS_BSW_00302] Limit exported information	SWS_Can_00037
[SRS_BSW_00328] Avoid duplication of code	Implementation requirement Fulfilled e.g. by defining one Can module that controls multiple channels
[SRS_BSW_00312] Shared code shall be reentrant	SWS_Can_00214, SWS_Can_00231, SWS_Can_00232, SWS_Can_00233
[SRS_BSW_00006] Platform independency	Chapter 1
[SRS_BSW_00439] Declaration of interrupt handlers and ISRs	not applicable
[SRS_BSW_00448] Module SWS shall not contain requirements from Other Modules	All chapters of this document containing SWS items
[SRS_BSW_00449] BSW Service APIs used by Autosar Application Software shall return a Std_ReturnType	not applicable
[SRS_BSW_00357] Standard API return type	not used
[SRS_BSW_00377] Module Specific API return type	SWS_Can_00039
[SRS_BSW_00304] AUTOSAR integer data types	standard integer data types are used
[SRS_BSW_00355] Do not redefine AUTOSAR integer data types	no redefined integer types in 8.2
[SRS_BSW_00378] AUTOSAR boolean type	not applicable (not used)
[SRS_BSW_00306] Avoid direct use of compiler and platform specific keywords	SWS_Can_00079
[SRS_BSW_00308] Definition of global data	SWS_Can_00079
[SRS_BSW_00309] Global data with read-only constraint	SWS_Can_00079
[SRS_BSW_00371] Do not pass function pointers via API	Chapter 0 (function definitions)
[SRS_BSW_00358] Return type of init() functions	SWS_Can_00223
[SRS_BSW_00414] Parameter of init function	SWS_Can_00223
[SRS_BSW_00376] Return type and parameters of main processing functions	SWS_Can_00031
[SRS_BSW_00359] Return type of callback functions	not applicable (no callback functions implemented in Can module)
[SRS_BSW_00360] Parameters of callback functions	no callbacks implemented in Can module
[SRS_BSW_00440] Function prototype for callback functions of AUTOSAR Services	not applicable
[SRS_BSW_00329] Avoidance of generic interfaces	No generic interface used. Still content of functions might be configuration dependent. Scope of function is always defined
[SRS_BSW_00330] Usage of macros instead of functions	SWS_Can_00079
[SRS_BSW_00331] Separation of error and status values	SWS_Can_00104, SWS_Can_00039
[BSW00443] Enabling / disabling defensive behavior of BSW	not applicable
[BSW00444] Error reporting and logging for defensive behavior of BSW	not applicable
[BSW00445] Protection against untimely call of BSW initialization	not applicable
[BSW00446] Protection against untimely call of BSW de-initialization	not applicable

[SRS_BSW_00009], [SRS_BSW_00401], [SRS_BSW_00172], [SRS_BSW_00010], [SRS_BSW_00333], [SRS_BSW_00374], [SRS_BSW_00379], [SRS_BSW_00003], [SRS_BSW_00318], [SRS_BSW_00321], [SRS_BSW_00341], [SRS_BSW_00334]	Software Documentation Requirements are not covered in the CAN Driver SWS

Document: AUTOSAR requirements on Basic Software, cluster SPAL (general SPAL requirements) [3]

Requirement	***Satisfied by***
[SRS_SPAL_12263] Object code compatible configuration concept	SWS_Can_00021
[SRS_SPAL_12056] Configuration of notification mechanisms	SWS_Can_00235
[SRS_SPAL_12267] Configuration of wake-up sources	ECUC_Can_00330
[SRS_SPAL_12057] Driver module initialization	SWS_Can_00245, SWS_Can_00246
[SRS_SPAL_12125] Initialization of hardware resources	SWS_Can_00053
[SRS_SPAL_12163] Driver module de-initialization	not applicable (decision in JointMM Meeting: no de-initialization for drivers that don't need to store non volatile information)
[SRS_SPAL_12461] Responsibility for register initialization	SWS_Can_00407
[SRS_SPAL_12462] Provide settings for register initialization	not applicable (Software Documentation Requirements are not covered in the CAN Driver SWS)
[SRS_SPAL_12463] Combine and forward settings for register initialization	SWS_Can_00024
[SRS_SPAL_12068] MCAL initialization sequence	not applicable (requirement on ECU state manager)
[SRS_SPAL_12069] Wake-up notification of ECU State Manager	SWS_Can_00271, SWS_Can_00364
[SRS_SPAL_00157] Notification mechanisms of drivers and handlers	SWS_Can_00026, SWS_Can_00028, SWS_Can_00031, SWS_Can_00108, SWS_Can_00109, SWS_Can_00112
[SRS_SPAL_12169] Control of operation mode	SWS_Can_00017
[SRS_SPAL_12063] Raw value mode	SWS_Can_00059, SWS_Can_00060
[SRS_SPAL_12075] Use of application buffers	SWS_Can_00011
[SRS_SPAL_12129] Resetting of interrupt flags	SWS_Can_00033
[SRS_SPAL_12064] Change of operation mode during running operation	not applicable
[SRS_SPAL_12448] Behavior after development error detection	SWS_Can_00091, SWS_Can_00089
[SRS_SPAL_12067] Setting of wake-up conditions	SWS_Can_00257
[SRS_SPAL_12077] Non-blocking implementation	SWS_Can_00371, SWS_Can_00372
[SRS_SPAL_12078] Runtime and memory efficiency	no effect on API definition implementation requirement
[SRS_SPAL_12092] Access to drivers	SWS_Can_00058
[SRS_SPAL_12265] Configuration data shall be kept constant	SWS_Can_00021 (stored in ROM -> implicitly constant)
[SRS_SPAL_12264] Specification of configuration items	Chapter 10

Document: AUTOSAR requirements on Basic Software, cluster CAN Driver [4]

Requirement	***Satisfied by***

[SRS_Can_01125] Data throughput read direction	not applicable (requirement affects complete COM stack and will not be broken down for the individual layers)
[SRS_Can_01126] Data throughput write direction	not applicable (requirement affects complete COM stack and will not be broken down for the individual layers)
[SRS_Can_01139] CAN controller specific initialization	SWS_Can_00062
[SRS_Can_01033] Basic Software Modules Requirements	see table above
[SRS_Can_01034] Hardware independent implementation	Chapter 1
[SRS_Can_01035] Multiple CAN controller support	Chapter 1
[SRS_Can_01036] CAN Identifier Length Configuration	ECUC_Can_00065
[SRS_Can_01037] Hardware Filter Configuration	ECUC_Can_00066, ECUC_Can_00325
[SRS_Can_01038] Bit Timing Configuration	ECUC_Can_00005, ECUC_Can_00073, ECUC_Can_00074, ECUC_Can_00075
[SRS_Can_01039] CAN Hardware Object Handle definitions	ECUC_Can_00324
[SRS_Can_01040] HW Transmit Cancellation configuration	ECUC_Can_00069
[SRS_Can_01058] Configuration of multiplexed transmission	ECUC_Can_00095
[SRS_Can_01062] Configuration of polling mode	SWS_Can_00007, ECUC_Can_00314, ECUC_Can_00317, ECUC_Can_00318, ECUC_Can_00319,
[SRS_Can_01135] Configuration of multiple TX Hardware Objects	SWS_Can_00100
[SRS_Can_01041] Can module Module Initialization	SWS_Can_00245, SWS_Can_00246
[SRS_Can_01042] Selection of static configuration sets	SWS_Can_00062
[SRS_Can_01043] Enable/disable Interrupts	SWS_Can_00049, SWS_Can_00050
[SRS_Can_01059] Data Consistency	SWS_Can_00011, SWS_Can_00012
[SRS_Can_01045] Reception Indication Service	SWS_Can_00279, SWS_Can_00396
[SRS_Can_01049] Dynamic transmission request service	SWS_Can_00212, SWS_Can_00213, SWS_Can_00214
[SRS_Can_01051] Transmit Confirmation	SWS_Can_00016
[SRS_Can_01053] CAN controller mode select	SWS_Can_00017
[SRS_Can_01054] Wake-up Notification	SWS_Can_00235, SWS_Can_00271, SWS_Can_00364
[SRS_Can_01132] Mixed mode for notification detection on CAN HW	SWS_Can_00099
[SRS_Can_01133] HW Transmit Cancellation Support	SWS_Can_00285, SWS_Can_00286, SWS_Can_00287, SWS_Can_00288, SWS_Can_00278, SWS_Can_00399, SWS_Can_00400
[SRS_Can_01134] Multiplexed Transmission	SWS_Can_00076, SWS_Can_00277, SWS_Can_00401, SWS_Can_00402, SWS_Can_00403
[SRS_Can_01055] Bus-off Notification	SWS_Can_00020, SWS_Can_00234
[SRS_Can_01060] no automatic bus-off recovery	SWS_Can_00272, SWS_Can_00273, SWS_Can_00274
[SRS_Can_01122] Support for wakeup during sleep transition	SWS_Can_00048
[SRS_Can_01147] No Remote Frame Support	SWS_Can_00236, SWS_Can_00237

7 Functional specification

On L-PDU transmission, the Can module writes the L-PDU in an appropriate buffer inside the CAN controller hardware.
See chapter 7.5 for closer description of L-PDU transmission.
On L-PDU reception, the Can module calls the RX indication callback function with ID, DLC and pointer to L-SDU as parameter.
See chapter 7.6 for closer description of L-PDU reception.
The Can module provides an interface that serves as periodical processing function, and which must be called by the Basic Software Scheduler module periodically.
Furthermore, the Can module provides services to control the state of the CAN controllers. Bus-off and Wake-up events are notified by means of callback functions.
The Can module is a Basic Software Module that accesses hardware resources. Therefore, it is designed to fulfill the requirements for Basic Software Modules specified in AUTOSAR_SRS_SPAL (see [3]).

[SWS_Can_00033] ⌈The Can module shall implement the interrupt service routines for all CAN Hardware Unit interrupts that are needed. ⌋(SRS_BSW_00164, SRS_SPAL_12129)

[SWS_Can_00419] ⌈The Can module shall disable all unused interrupts in the CAN controller.⌋()

[SWS_Can_00420] ⌈The Can module shall reset the interrupt flag at the end of the ISR (if not done automatically by hardware). ⌋()

Implementation hint: The Can module shall not set the configuration (i.e. priority) of the vector table entry.

[SWS_Can_00079] ⌈The Can module shall fulfill all design and implementation guidelines described in [11].⌋(SRS_BSW_00007, SRS_BSW_00306, SRS_BSW_00308, SRS_BSW_00309, SRS_BSW_00330)

7.1 Driver scope

One Can module provides access to one CAN Hardware Unit that may consist of several CAN controllers.

[SWS_Can_00077] ⌈For CAN Hardware Units of different type, different Can modules shall be implemented. ⌋(SRS_BSW_00347)

[SWS_Can_00284] ⌈In case several CAN Hardware Units (of same or different vendor) are implemented in one ECU the function names, and global variables of the Can modules shall be implemented such that no two functions with the same name are generated.⌋()

The naming convention is as follows:

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

SRS_BSW_00347 specifies the naming convention.

[SWS_Can_00385] 「The naming conventions shall be used only in that case, if multiple different CAN controller types on one ECU have to be supported. 」()

[SWS_Can_00386] 「If only one controller type is used, the original naming conventions without any <driver abbreviation> extensions are sufficient.」()
See [5] for description how several Can modules are handled by the CanIf module.

7.2 Driver State Machine

The Can module has a very simple state machine, with the two states CAN_UNINIT and CAN_READY. Figure 7.1 shows the state machine.

[SWS_Can_00103] 「After power-up/reset, the Can module shall be in the state CAN_UNINIT. 」(SRS_BSW_00406)

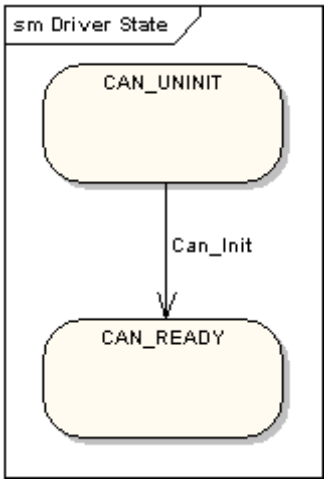


Figure 7-1

[SWS_Can_00246] 「The function Can_Init shall change the module state to CAN_READY, after initializing all controllers inside the HW Unit.」(SRS_SPAL_12057, SRS_Can_01041)

[SWS_Can_00245] 「The function Can_Init shall initialize all CAN controllers according to their configuration.」(SRS_SPAL_12057, SRS_Can_01041)

Each CAN controller must then be started separately by calling the function Can_SetControllerMode(CAN_T_START).
Implementation hint:
Hardware register settings that have impact on all CAN controllers inside the HW Unit can only be set in the function Can_Init.
Implementation hint:
The ECU State Manager module shall call Can_Init at most once during runtime.

7.3 CAN Controller State Machine

Each CAN controller has complex state machines implemented in hardware. For simplification, the number of states is reduced to the following four basic states in this description: UNINIT, STOPPED, STARTED and SLEEP.
For each CAN controller a corresponding 'software' state machine is implemented in the CanIf module [5] with the following states: CANIF_CS_UNINIT, CANIF_CS_STOPPED, CANIF_CS_STARTED and CANIF_CS_SLEEP. [5] shows the implementation of the software state machine. Any CAN hardware access is encapsulated by functions of the Can module, but the Can module does not memorize the state changes.
During a transition phase, the software controller state inside the CanIf module may differ from the hardware state of the CAN controller.
The Can module offers the services Can_Init, Can_SetBaudrate and Can_SetControllerMode. These services perform the necessary register settings that cause the required change of the hardware CAN controller state.
There are two possibilities for triggering state changes by external events:

- Bus-off event
- HW wakeup event

These events are indicated either by an interrupt or by a status bit that is polled in the Can_MainFunction_BusOff or Can_MainFunction_Wakeup.
The Can module does the register settings that are necessary to fulfill the required behavior (i.e. no hardware recovery in case of bus off).
Then it notifies the CanIf module with the corresponding callback function. The software state is then changed inside this callback function.
The Can module does not check for validity of state changes. It is the task of upper layer modules to trigger only transitions that are allowed in the current state. In case development errors are enabled, the Can module checks the transition. In case of wrong implementation by the upper layer module, the Can module raises the development error CAN_E_TRANSITION.
The Can module does not check the actual state before it performs Can_Write or raises callbacks.
During a transition phase - where the software controller state inside the CanIf module differs from the hardware state of the CAN controller – transmit might fail or be delayed because the hardware CAN controller is not yet participating on the bus. The Can module does not provide a notification for this case.

7.3.1 CAN Controller State Description

This chapter describes the required hardware behavior for the different SW states. The software state machine itself is implemented and described in the CanIf module. Please refer to [5] for the state diagram.

CAN controller state UNINIT

The CAN controller is not initialized. All registers belonging to the CAN module are in reset state, CAN interrupts are disabled. The CAN Controller is not participating on the CAN bus.

CAN controller state STOPPED

In this state the CAN Controller is initialized but does not participate on the bus. In addition, error frames and acknowledges must not be sent.
(Example: For many controllers entering an ‘initialization’-mode causes the controller to be stopped.)

CAN controller state STARTED

The controller is in a normal operation mode with complete functionality, which means it participates in the network. For many controllers leaving the ‘initialization’-mode causes the controller to be started.

CAN controller state SLEEP

The hardware settings only differ from state STOPPED for CAN hardware that support a sleep mode (wake-up over CAN bus directly supported by CAN hardware).

[SWS_Can_00257] 「When the CAN hardware supports sleep mode and is triggered to transition into SLEEP state, the Can module shall set the controller to the SLEEP state from which the hardware can be woken over CAN Bus.」(SRS_SPAL_12067)

[SWS_Can_00258] 「When the CAN hardware does not support sleep mode and is triggered to transition into SLEEP state, the Can module shall emulate a logical SLEEP state from which it returns only, when it is triggered by software to transition into STOPPED state.」()

[SWS_Can_00404] 「The CAN hardware shall remain in state STOPPED, while the logical SLEEP state is active.」()

7.3.2 CAN Controller State Transitions

A state transition is triggered by software with the function Can_SetControllerMode with the required transition as parameter. A successful state transition triggered by software is notified by the callback function (CanIf_ControllerModeIndication). The monitoring whether the requested state is achieved is part of an upper layer module and is not part of the Can module.
Some transitions are triggered by events on the bus (hardware). These transitions cause a notification by means of a callback function (CanIf_ControllerBusOff, EcuM_CheckWakeup).
Plausibility checks for state transitions are only performed with development error detection switched on. The behavior for invalid transitions in production code is undefined. Figure 7-2 shows all valid state transitions.

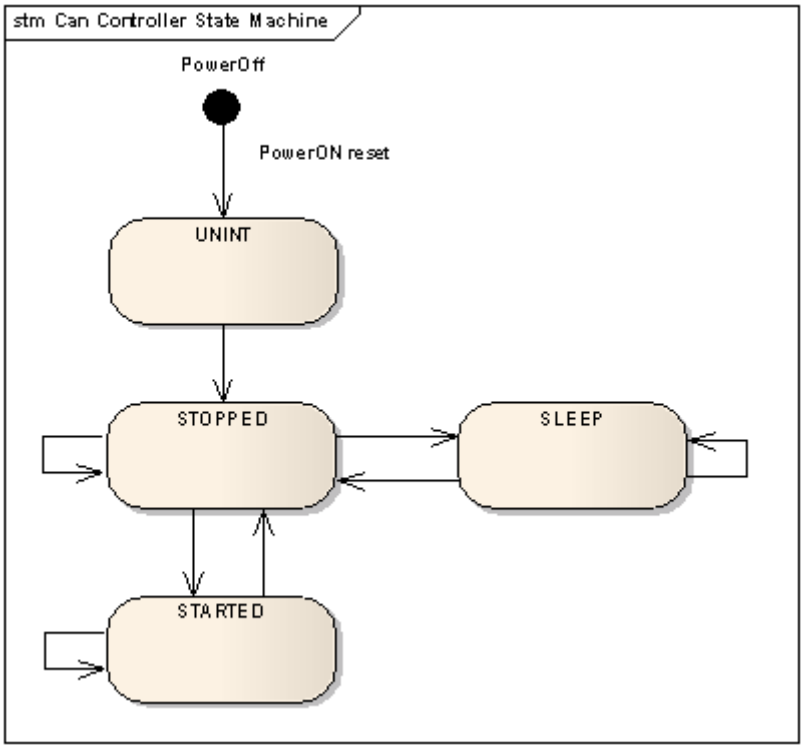


Figure 7-2

7.3.3 State transition caused by function Can_Init

- UNINIT → STOPPED (for all controllers in HW unit)
- software triggered by the function call Can_Init
- does configuration for all CAN controllers inside HW Unit

All control registers are set according to the static configuration.

[SWS_Can_00259] 「The function Can_Init shall set all CAN controllers in the state STOPPED.」()

When the function Can_Init is entered and the Can module is not in state CAN_UNINIT or the CAN controllers are not in state UNINIT, it shall raise the error CAN_E_TRANSITION (Compare to SWS_Can_00174 and SWS_Can_00408).

7.3.4 State transition caused by function Can_SetBaudrate

- STOPPED -> STOPPED; SLEEP -> SLEEP; STARTED -> STARTED
- software triggered by the function call Can_SetBaudrate
- changes the CAN controller configuration

CAN controller registers are set according to the static configurations.

[SWS_Can_00256] ⌈If the call of Can_SetBaudrate will cause a re-initialization of the CAN Controller: The function Can_SetBaudrate shall return E_NOT_OK if the controller is not in state STOPPED and raise the error CAN_E_TRANSITION if development error detection for the Can module is enabled.⌋()

[SWS_Can_00260] ⌈If re-initialization is necessary the function Can_SetBaudrate shall maintain the CAN controller in the state STOPPED.⌋()

[SWS_Can_00422] ⌈If re-initialization is necessary the function Can_SetBaudrate shall ensure that any settings that will cause the CAN controller to participate in the network are not set.⌋()

7.3.5 State transition caused by function Can_SetControllerMode

The software can trigger a CAN controller state transition with the function Can_SetControllerMode. Depending on the CAN hardware, a change of a register setting to transition to a new CAN controller state may take over only after a delay. The Can module notifies the upper layer (CanIf_ControllerModeIndication) after a successful state transition about the new state. The monitoring whether the requested state is achieved is part of an upper layer module and is not part of the Can module.

[SWS_Can_00370] ⌈The function Can_Mainfunction_Mode shall poll a flag of the CAN status register until the flag signals that the change takes effect and notify the upper layer with function CanIf_ControllerModeIndication about a successful state transition.⌋()

[SWS_Can_00371] ⌈This polling shall take the maximum time of `CanTimeoutDuration` for blocking function and thus the polling time is limited.⌋(SRS_SPAL_12077)

[SWS_Can_00398] ⌈The function Can_SetControllerMode shall use the system service GetCounterValue for timeout monitoring to avoid blocking functions.⌋()

[SWS_Can_00372] ⌈In case the flag signals that the change takes no effect and the maximum time `CanTimeoutDuration` is elapsed, the function Can_SetControllerMode shall be left and the function Can_Mainfunction_Mode shall continue to poll the flag.⌋(SRS_SPAL_12077)

[SWS_Can_00373] ⌈The function Can_Mainfunction_Mode shall call the function CanIf_ControllerModeIndication to notify the upper layer about a successful state transition of the CAN controller, in case the state transition was triggered by function Can_SetControllerMode.⌋()

State transition caused by function Can_SetControllerMode(CAN_T_START)

- STOPPED → STARTED

- software triggered

[SWS_Can_00261] ⌈The function Can_SetControllerMode(CAN_T_START) shall set the hardware registers in a way that makes the CAN controller participating on the network.⌋()

[SWS_Can_00262] ⌈The function Can_SetControllerMode(CAN_T_START) shall wait for limited time until the CAN controller is fully operational. Compare to SWS_Can_00371.⌋()

Transmit requests that are initiated before the CAN controller is operational get lost. The only indicator for operability is the reception of TX confirmations or RX indications. The sending entities might get a confirmation timeout and need to be able to cope with that.

[SWS_Can_00409] ⌈When the function Can_SetControllerMode(CAN_T_START) is entered and the CAN controller is not in state STOPPED it shall detect a invalid state transition (Compare to SWS_Can_00200).⌋()

State transition caused by function Can_SetControllerMode(CAN_T_STOP)

- STARTED → STOPPED
- software triggered

[SWS_Can_00263] ⌈The function Can_SetControllerMode(CAN_T_STOP) shall set the bits inside the CAN hardware such that the CAN controller stops participating on the network.⌋()

[SWS_Can_00264] ⌈The function Can_SetControllerMode(CAN_T_STOP) shall wait for a limited time until the CAN controller is really switched off. Compare to SWS_Can_00371.⌋()

[SWS_Can_00282] ⌈The function Can_SetControllerMode(CAN_T_STOP) shall cancel pending messages. ⌋()

[SWS_Can_00283] ⌈The function Can_SetControllerMode(CAN_T_STOP) shall not call a cancellation notification.⌋()

Hint: Even if pending messages are cancelled by the function Can_SetControllerMode(CAN_T_STOP), there are hardware restrictions and racing problems. So it cannot be guaranteed if the cancelled messages are still processed by the hardware or not.

[SWS_Can_00410] ⌈When the function Can_SetControllerMode(CAN_T_STOP) is entered and the CAN controller is neither in state STARTED nor in state STOPPED, it shall detect a invalid state transition (Compare to SWS_Can_00200).⌋()

State transition caused by function Can_SetControllerMode(CAN_T_SLEEP)

- STOPPED → SLEEP
- software triggered

[SWS_Can_00265] ⌈The function Can_SetControllerMode(CAN_T_SLEEP) shall set the controller into sleep mode.⌋()

[SWS_Can_00266] ⌈If the CAN HW does support a sleep mode, the function Can_SetControllerMode(CAN_T_SLEEP) shall wait for a limited time until the CAN controller is in SLEEP state and it is assured that the CAN hardware is wake able. Compare to SWS_Can_00371.⌋()

[SWS_Can_00290] ⌈If the CAN HW does not support a sleep mode, the function Can_SetControllerMode(CAN_T_SLEEP) shall set the CAN controller to the logical sleep mode.⌋()

[SWS_Can_00405] ⌈This logical sleep mode shall left only, if function Can_SetControllerMode(CAN_T_WAKEUP) is called.⌋()

[SWS_Can_00411] ⌈When the function Can_SetControllerMode(CAN_T_SLEEP) is entered and the CAN controller is neither in state STOPPED nor in state SLEEP, it shall detect a invalid state transition (Compare to SWS_Can_00200).⌋()

State transition caused by function Can_SetControllerMode(CAN_T_WAKEUP)

- SLEEP → STOPPED
- software triggered

[SWS_Can_00267] ⌈If the CAN HW does not support a sleep mode, the function Can_SetControllerMode(CAN_T_WAKEUP) shall return from the logical sleep mode, but have no effect to the CAN controller state (as the controller is already in stopped state).⌋()

[SWS_Can_00268] ⌈The function Can_SetControllerMode(CAN_T_WAKEUP) shall wait for a limited time until the CAN controller is in STOPPED state. Compare to SWS_Can_00371.⌋()

[SWS_Can_00412] ⌈When the function Can_SetControllerMode(CAN_T_WAKEUP) is entered and the CAN controller is neither in state SLEEP nor in state STOPPED, it shall detect a invalid state transition (Compare to SWS_Can_00200).⌋()

7.3.6 State transition caused by Hardware Events

State transition caused by Hardware Wakeup (triggered by wake-up event from CAN bus)

- SLEEP → STOPPED

- triggered by incoming L-PDUs
- The ECU Statemanager module is notified with the function EcuM_CheckWakeup

This state transition will only occur when sleep mode is supported by hardware.

[SWS_Can_00270] ⌈On hardware wakeup (triggered by a wake-up event from CAN bus), the CAN controller shall transition into the state STOPPED.⌋()

[SWS_Can_00271] ⌈On hardware wakeup (triggered by a wake-up event from CAN bus), the Can module shall call the function EcuM_CheckWakeup either in interrupt context or in the context of Can_MainFunction_Wakeup.⌋(SRS_BSW_00375, SRS_SPAL_12069, SRS_Can_01054)

[SWS_Can_00269] ⌈The Can module shall not further process the L-PDU that caused a wake-up.⌋()

[SWS_Can_00048] ⌈In case of a CAN bus wake-up during sleep transition, the function Can_SetControllerMode(CAN_T_WAKEUP) shall return CAN_NOT_OK.⌋(SRS_Can_01122)

State transition caused by Bus-Off (triggered by state change of CAN controller)

[SWS_Can_00020] ⌈

- STARTED → STOPPED
- triggered by hardware if the CAN controller reaches bus-off state
- The CanIf module is notified with the function CanIf_ControllerBusOff after STOPPED state is reached.⌋(SRS_Can_01055)

[SWS_Can_00272] ⌈After bus-off detection, the CAN controller shall transition to the state STOPPED and the Can module shall ensure that the CAN controller doesn't participate on the network anymore. ⌋(SRS_Can_01060)

[SWS_Can_00273] ⌈After bus-off detection, the Can module shall cancel still pending messages without raising a cancellation notification. ⌋(SRS_Can_01060)

[SWS_Can_00274] ⌈The Can module shall disable or suppress automatic bus-off recovery.⌋(SRS_Can_01060)

7.4 Can module/Controller Initialization

The ECU State Manager module shall initialize the Can module during startup phase by calling the function Can_Init before using any other functions of the Can module.

[SWS_Can_00250] ⌈The function Can_Init shall initialize:

- static variables, including flags,
- Common setting for the complete CAN HW unit

- CAN controller specific settings for each CAN controller」(SRS_BSW_00101)

[SWS_Can_00053]「Can_Init shall not change registers of CAN controller Hardware resources that are not used. 」(SRS_SPAL_12125)
The Can module shall apply the following rules regarding initialization of controller registers:

- **[SWS_Can_00407]**「If the hardware allows for only one usage of the register, the Can module implementing that functionality is responsible initializing the register.
- If the register can affect several hardware modules and if it is an I/O register it shall be initialized by the PORT driver.
- If the register can affect several hardware modules and if it is not an I/O register it shall be initialized by the MCU driver.
- One-time writable registers that require initialization directly after reset shall be initialized by the startup code.
- All other registers shall be initialized by the startup code.」(SRS_SPAL_12461)

[SWS_Can_00056]「Post-Build configuration elements that are marked as 'multiple' ('M' or 'x') in chapter 10 can be selected by passing the pointer 'Config' to the init function of the module. 」()

[SWS_Can_00062]「If re-initialization is necessary the function Can_SetBaudrate shall re-initialize the CAN controller and the controller specific settings.」(SRS_Can_01139, SRS_Can_01042)
If re-initialization is necessary the CanIf module must first set the CAN controller in STOPPED state before Can_SetBaudrate can be invoked.
[SWS_Can_00255]「The function Can_SetBaudrate shall only affect register areas that contain specific configuration for a single CAN controller. 」()
[SWS_Can_00021]「The desired CAN controller configuration can be selected with the parameter Config. 」(SRS_BSW_00344, SRS_BSW_00404, SRS_BSW_00405, SRS_SPAL_12263, SRS_SPAL_12265)
[SWS_Can_00291]「Config is a pointer into an array of implementation specific data structure stored in ROM. The different controller configuration sets are located as data structures in ROM.」(SRS_BSW_00438)
The possible values for Config are provided by the configuration description (see chapter 10).
The Can module configuration defines the global CAN HW Unit settings and references to the default CAN controller configuration sets.

7.5 L-PDU transmission

On L-PDU transmission, the Can module converts the L-PDU contents ID and DLC to a hardware specific format (if necessary) and triggers the transmission.

[SWS_Can_00059] ⌈Data mapping by CAN to memory is defined in a way that the CAN data byte which is sent out first is array element 0, the CAN data byte which is sent out last is array element 7.⌋(SRS_SPAL_12063)

[SWS_Can_00427] ⌈If the presentation inside the CAN Hardware buffer differs from AUTOSAR definition, the Can module must provide an adapted SDU-Buffer for the upper layers.⌋()

[SWS_Can_00100] ⌈Several TX hardware objects with unique HTHs may be configured. The CanIf module provides the HTH as parameter of the TX request. See Figure 7-3 for a possible configuration.⌋(SRS_Can_01135)

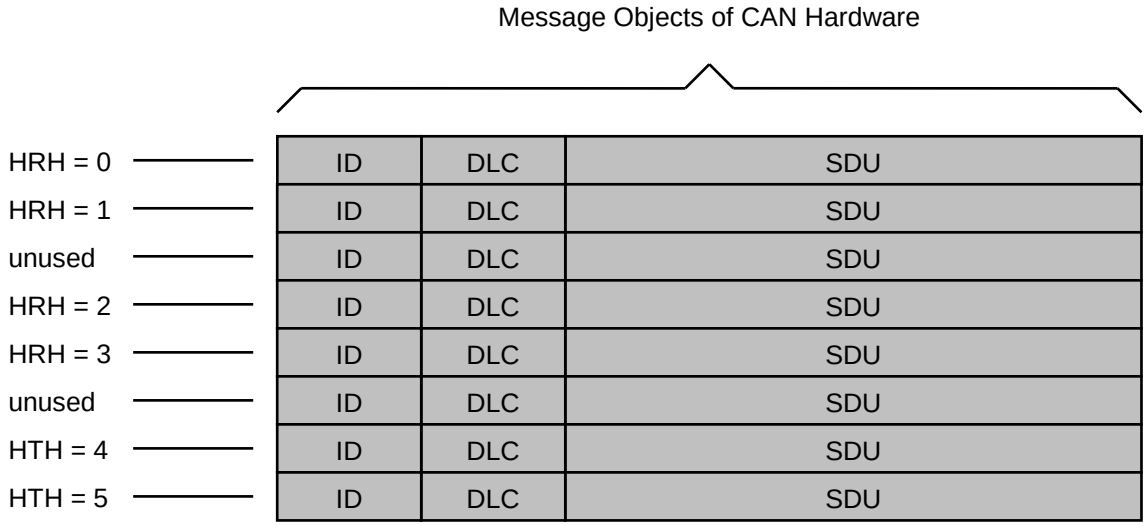


Figure 7-3: Example of assignment of HTHs and HRHs to the Hardware Objects. The numbering of HTHs and HRHs are implementation specific. The chosen numbering is only an example.

[SWS_Can_00276] ⌈The function Can_Write shall store the swPduHandle that is given inside the parameter PduInfo until the Can module calls the CanIf_TxConfirmation for this request where the swPduHandle is given as parameter. ⌋()

The feature of SWS_Can_00276 is used to reduce time for searching in the CanIf module implementation.

[SWS_Can_00016] ⌈The Can module shall call CanIf_TxConfirmation to indicate a successful transmission. It shall either called by the TX-interrupt service routine of the corresponding HW resource or inside the Can_MainFunction_Write in case of polling mode.⌋(SRS_Can_01051)

7.5.1 Priority Inversion

To prevent priority inversion two mechanisms are necessary: multiplexed transmission and hardware cancellation (see chapter 2.1).

7.5.1.1 Multiplexed Transmission

[SWS_Can_00277] ⌈The Can module shall allow that the functionality “Multiplexed Transmission” is statically configurable (ON | OFF) at pre-compile time.⌋(SRS_Can_01134)

[SWS_Can_00401] ⌈Several transmit hardware objects (defined by "CanHwObjectCount") shall be assigned by one HTH to represent one transmit entity to the upper layer.⌋(SRS_Can_01134)

[SWS_Can_00402] ⌈The Can module shall support multiplexed transmission mechanisms for devices where either

- Multiple transmit hardware objects, which are grouped to a transmit entity can be filled over the same register set, and the microcontroller stores the L-PDU into a free buffer autonomously,

or

- The Hardware provides registers or functions to identify a free transmit hardware object within a transmit entity.⌋(SRS_Can_01134)

[SWS_Can_00403] ⌈The Can module shall support multiplexed transmission for devices, which send L-PDUs in order of L-PDU priority.⌋(SRS_Can_01134)

Note: Software emulation of priority handling should be avoided, because the overhead would void the advantage of the multiplexed transmission.

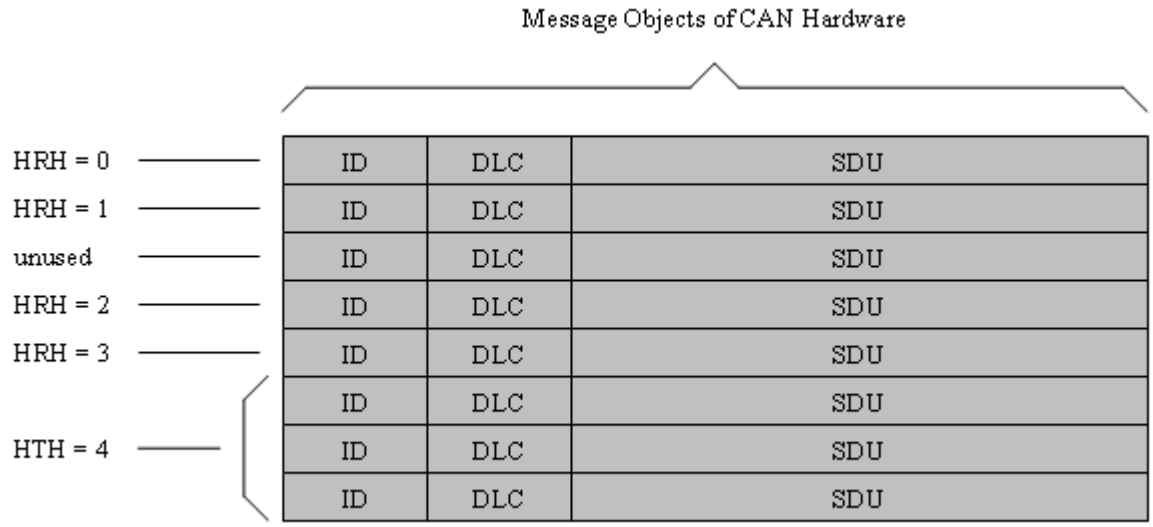


Figure 7-4: Example of assignment of HTHs and HRHs to the Hardware Objects with multiplexed transmission. The numbering of HTHs and HRHs are implementation specific. The chosen numbering is only an example.

7.5.1.2 Transmit Cancellation

For some applications, it is required to transmit always the newest data on the bus. L-PDUs which are pending in the transmit buffer from the previous transmit cycle must be replaced by an L-PDU of current transmit cycle. This requirement is supported by cancellation of pending L-PDUs with identical priority. However, cancellation and replacement of an L-PDU with identical priority can lead to priority inversion, which is in conflict with the requirement to prevent priority inversion. To satisfy both requirements, the configuration parameter *CanIdenticalIdCancellation* enables/disables the cancellation of L-PDUs with identical priority.

[SWS_Can_00278] ⌈The Can module shall allow that the functionality “Transmit Cancellation” is statically configurable (ON | OFF) at pre-compile time.⌋(SRS_Can_01133)

The complete cancellation sequence is described in the CanIf module [5].

[SWS_Can_00432] ⌈The Can module shall allow that the cancellation of pending L-PDUs with identical priority is statically configurable at pre-compile time by parameter *CanIdenticalIdCancellation.*⌋()

[SWS_Can_00285] ⌈Transmit cancellation may only be used when transmit buffers are enabled inside the CanIf module.⌋(SRS_Can_01133)

[SWS_Can_00286] ⌈The Can module shall initiate a cancellation, when the hardware transmit object assigned by a HTH is busy and an L-PDU with higher priority is requested to be transmitted.⌋(SRS_Can_01133)

[SWS_Can_00433] ⌈The Can module shall initiate a cancellation, when the hardware transmit object assigned by a HTH is busy, an L-PDU with identical priority is requested to be transmitted and *CanIdenticalIdCancellation* is enabled.⌋()

The following two items are valid, in case multiplexed transmission functionality is enabled and several hardware transmit objects are assigned by one HTH:

[SWS_Can_00399] ⌈The Can module shall initiate a cancellation of the L-PDU with the lowest priority, when all hardware transmit objects assigned by the HTH are busy and an L-PDU with a higher priority is requested to be transmitted.⌋(SRS_Can_01133)

[SWS_Can_00400] ⌈The Can module shall initiate a cancellation, when one of the hardware transmit objects assigned by the HTH is busy, an L-PDU with identical priority is requested to be transmitted and *CanIdenticalIdCancellation* is enabled.⌋(SRS_Can_01133)

The incoming request is also rejected because the cancellation is asynchronous.

[SWS_Can_00287] ⌈The Can module shall raise a notification when the cancellation was successful by calling the function CanIf_CancelTxConfirmation.⌋(SRS_Can_01133)

[SWS_Can_00288] ⌈The TX request for the new L-PDU shall be repeated by the CanIf module, inside the notification function CanIf_CancelTxConfirmation. ⌋(SRS_Can_01133)
Implementation note:
For sequence relevant streams the sender must assure that the next transmit request for the same CAN ID is only initiated after the last request was confirmed.

7.5.2 Transmit Data Consistency

[SWS_Can_00011] ⌈The Can module shall directly copy the data from the upper layer buffers. It is the responsibility of the upper layer to keep the buffer consistent until return of function call (Can_Write).⌋(SRS_SPAL_12075, SRS_Can_01059)

7.6 L-PDU reception

[SWS_Can_00279] ⌈On L-PDU reception, the Can module shall call the RX indication callback function CanIf_RxIndication with ID, Hoh, ControllerId in parameter Mailbox, and the DLC and pointer to the L-SDU buffer in parameter PduInfoPtr.⌋(SRS_Can_01045)

[SWS_Can_00423] ⌈In case of an Extended CAN frame, the Can module shall convert the ID to a standardized format since the Upper layer (CANIF) does not know whether the received CAN frame is a Standard CAN frame or Extended CAN frame. In case of an Extended CAN frame, MSB of a received CAN frame ID needs to be made as '1' to mark the received CAN frame as Extended.⌋()

Note: CanDrv does not indicate whether the received message is a conventional CAN frame or a CAN FD frame.

[SWS_Can_00396] ⌈The RX-interrupt service routine of the corresponding HW resource or the function Can_MainFunction_Read in case of polling mode shall call the callback function CanIf_RxIndication.⌋(SRS_Can_01045)

[SWS_Can_00060] ⌈Data mapping by CAN to memory is defined in a way that the CAN data byte which is received first is array element 0, the CAN data byte which is received last is array element 7.
If the presentation inside the CAN Hardware buffer differs from AUTOSAR definition, the Can module must provide an adapted SDU-Buffer for the upper layers.⌋(SRS_SPAL_12063)

7.6.1 Receive Data Consistency

To prevent loss of received messages, some controllers support a FIFO built from a set of hardware objects, while on other controllers it is possible to

configure another hardware object with the same properties that works as a shadow buffer and steps in when the main object is busy.

[**SWS_CAN_00489**]⌈ The CAN driver shall support controllers which implement a hardware FIFO. The size of the FIFO is configured via "CanHwObjectCount".
⌋()

[**SWS_CAN_00490**]⌈ Controllers that do not support a hardware FIFO often provide the capabilities to implement a shadow buffer mechanism, where additional hardware objects take over when the primary hardware object is busy. The number of hardware objects is configured via "CanHwObjectCount".⌋()

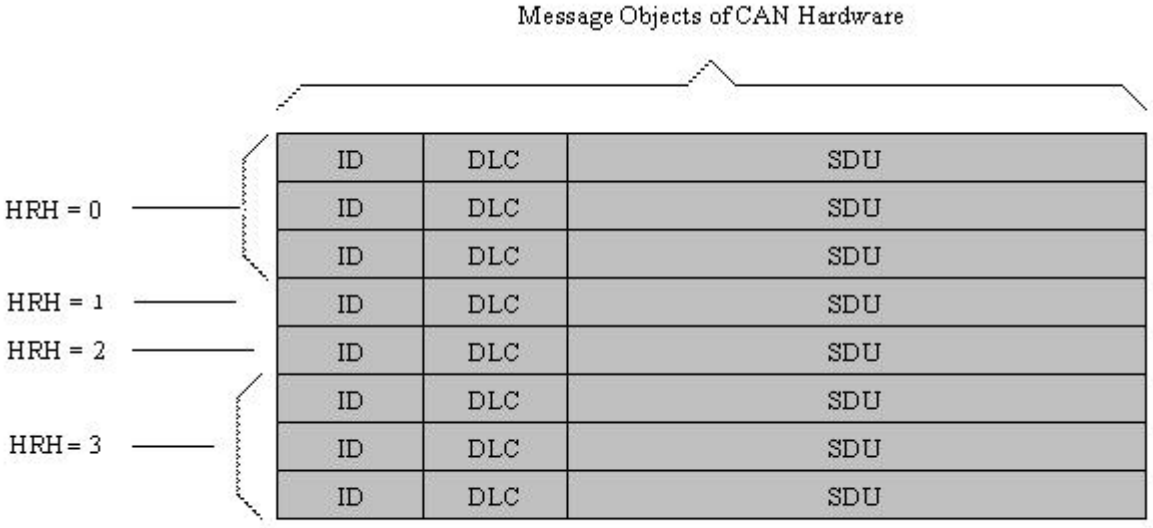


Figure 7-5: Example of assignment of same HRHs to multiple Hardware Objects The chosen numbering is only an example.

[SWS_Can_00299] ⌈The Can module shall copy the L-SDU in a shadow buffer after reception, if the RX buffer cannot be protected (locked) by CAN Hardware against overwriting by a newly received message.⌋()

[SWS_Can_00300] ⌈The Can module shall copy the L-SDU in a shadow buffer, if the CAN Hardware is not globally accessible.⌋()

The complete RX processing (including copying to destination layer, e.g. COM) is done in the context of the RX interrupt or in the context of the Can_MainFunction_Read.

[SWS_Can_00012] ⌈The Can module shall guarantee that neither the ISRs nor the function Can_MainFunction_Read can be interrupted by itself. The CAN hardware (or shadow) buffer is always consistent, because it is written and read in sequence in exactly one function that is never interrupted by itself.⌋(SRS_Can_01059)

If the CAN hardware cannot be configured to lock the RX hardware object after reception (hardware feature), it could happen that the hardware buffer is overwritten

by a newly arrived message. In this case, the CAN controller detects an “overwrite” event, if supported by hardware.
If the CAN hardware can be configured to lock the RX hardware object after reception, it could happen that the newly arrived message cannot be stored to the hardware buffer. In this case, the CAN controller detects an “overrun” event, if supported by hardware.

[SWS_Can_00395] 「If the development error detection for the Can module is enabled, the Can module shall raise the error CAN_E_DATALOST in case of “overwrite” or “overrun” event detection.」()

Implementation Hint:
The system designer shall assure that the runtime for message reception (interrupt driven or polling) correlates with the fasted possible reception in the system.

7.7 Wakeup concept

The Can module handles wakeups that can be detected by the Can controller itself and not via the Can transceiver. There are two possible scenarios: wakeup by interrupt and wakeup by polling.
For wakeup by interrupt, an ISR of the Can module is called when the hardware detects the wakeup.

[SWS_Can_00364] 「If the ISR for wakeup events is called, it shall call EcuM_CheckWakeup in turn. The parameter passed to EcuM_CheckWakeup shall be the ID of the wakeup source referenced by the CanWakeupSourceRef configuration parameter.」(SRS_BSW_00375, SRS_SPAL_12069, SRS_Can_01054)

The ECU State Manager will then set up the MCU and call the Can module back via the Can Interface, resulting in a call to Can_CheckWakeup.
When wakeup events are detected by polling, the ECU State Manager will cyclically call Can_CheckWakeup via the Can Interface as before. In both cases, Can_CheckWakeup will check if there was a wakeup detected by a Can controller and return the result. The CAN driver will then inform the ECU State Manager of the wakeup event via EcuM_SetWakeupEvent.
The wakeup validation to prevent false wakeup events, will be done by the ECU State Manager and the Can Interface afterwards and without any help from the Can module.
For a general description of the wakeup mechanisms and wakeup sequence diagrams refer to Specification of ECU State Manager [7].

7.8 Notification concept

The Can module offers only an event triggered notification interface to the CanIf module. Each notification is represented by a callback function.

[SWS_Can_00099] 「The hardware events may be detected by an interrupt or by polling status flags of the hardware objects. The configuration possibilities regarding polling is hardware dependent (i.e. which events can be polled, which events need to be polled), and not restricted by this standard. 」(SRS_Can_01132)

[SWS_Can_00007] 「It shall be possible to configure the driver such that no interrupts at all are used (complete polling). 」(SRS_Can_01062)

The configuration of what is and is not polled by the Can module is internal to the driver, and not visible outside the module. The polling is done inside the CAN main functions (Can_MainFunction_xxx). Also the polled events are notified by the appropriate callback function. Then the call context is not the ISR but the CAN main function. The implementation of all callback functions shall be done as if the call context was the ISR.
For further details see also description of the CAN main functions Can_MainFunction_Read, Can_MainFunction_Write, Can_MainFunction_BusOff and Can_MainFunction_Wakeup.

7.9 Reentrancy issues

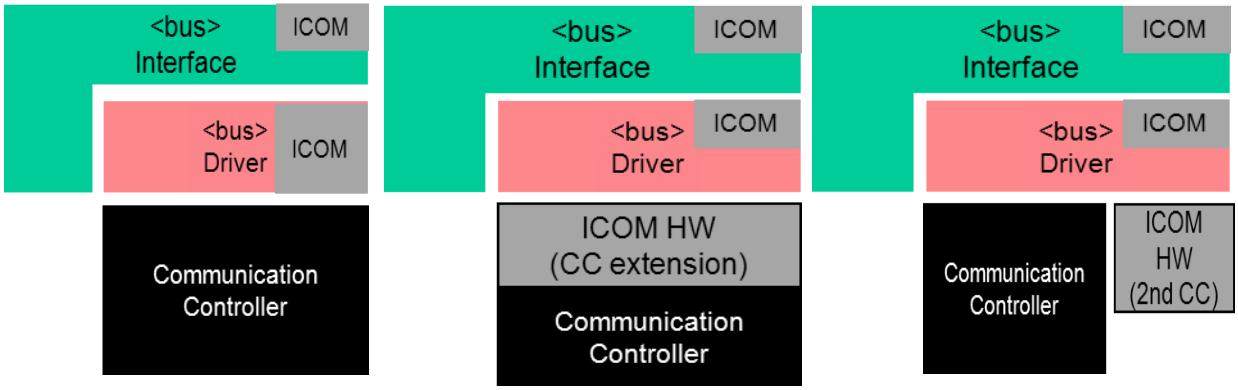
A routine must satisfy the following conditions to be reentrant:

- It uses all shared variables in an atomic way, unless each is allocated to a specific instance of the function.
- It does not call non-reentrant functions.
- It does not use the hardware in a non-atomic way.

Transmit requests are simply forwarded by the CanIf module inside the function CanIf_Transmit.
The function CanIf_Transmit is re-entrant. Therefore the function Can_Write needs to be implemented thread-safe (for example by using mutexes):
Further (preemptive) calls will return with CAN_BUSY when the write can't be performed re-entrant. (example: write to different hardware TX Handles allowed, write to same TX Handles not allowed)
In case of CAN_BUSY the CanIf module queues that request. (same behavior as if all hardware objects are busy).
Can_EnableCanInterrupts and Can_DisableCanInterrupts may be called inside re-entrant functions. Therefore these functions also need to be reentrant.
All other services don't need to be implemented as reentrant functions.
The CAN main functions (i.e. Can_MainFunction_Read) shall not be interrupted by themselves. Therefore these CAN main functions are not reentrant.

7.10 Pretended Networking

Optimizing energy efficiency is becoming increasingly important in all automotive domains since energy consumption has direct impact on fuel consumption, CO2 emissions, and range of hybrid or all electric vehicles. The concept of *Pretended Networking* has a high potential for energy reduction on ECU level.



(a) ICOM Software implementation: ICOM features are exclusively implemented in software, reusing the existing Communication Controller

(b) ICOM is a functional hardware extension of the existing Communication Controller

(c) ICOM is a separate hardware component with its own Communication Controller

Figure 7-6: Possible ICOM implementations

The use of *Intelligent Communication Controllers* (ICOM) is planned to support those features (no specific hardware implementation mandatory). If some or all of the functionality of an ECU is temporarily not required, e.g., based on the vehicle state, the ECU can enter a “Pretended Networking” mode. In this mode, the MCU and/or peripherals are switched into a low-power mode. Only the ICOM and the connected transceivers stay active. The ICOM generates a wakeup event, caused by, e.g. a received bus message or local event, when the ECU needs to resume operation.
Depending on the ICOM implementation, message ID and payload of received messages may either be evaluated and filtered completely in hardware, require a callback mechanism in software, or be a combination of both. As shown in Figure 7-5, the ECU implementation can be divided into three possible variants - a software approach with re-use of state-of-the-art communication controllers (a). Approach (b) depicts a functional hardware extension of state-of-the-art communication controller and approach (c) shows a hardware variant with a 2nd extended communication controller for wakeup handling. All variants and ICOM implementations shall be supported by Pretended Networking.

Depending on the hardware implementation, the ICOM is also able to send messages. By using the ICOM to continue to send, e.g., status messages, other nodes that rely on that message are not affected by an ECU in Pretended Networking mode.

Furthermore, Pretended Networking aims at reducing wakeup response time, i.e., the time between a wakeup event and valid behavior of an ECU. By using the ICOM to save relevant messages during activated Pretended Networking mode, the application has access to the last valid signal values directly after resuming operation. Therefore, the ECU can immediately respond to a user request after wakeup and does not have to wait until the according message is received again.

7.10.1 Support Pretended Networking mode handling

[SWS_CAN_00497] ⌈The CAN driver shall deactivate Pretended Networking after initialization of the CAN controller.⌋()

Activation of Pretended Networking:

[SWS_CAN_00462]⌈ Pretended Networking shall be activated by calling Can_SetIcomConfiguration with a configuration ID greater 0. ⌋()

[SWS_CAN_00463]⌈CAN driver is responsible to check the configuration parameters if a new configuration shall be applied for CAN controller and ICOM. ⌋()

[SWS_CAN_00464]⌈CAN driver is responsible to perform reconfiguration of the CAN controller (incl. ICOM). ⌋()

[SWS_CAN_00465]⌈CAN driver is responsible to notify CAN interface if activation of Pretended Networking was successful. ⌋()

[SWS_CAN_00466]⌈If activation of Pretended Networking was successful the CAN driver shall store the information that ICOM is activated. ⌋()

[SWS_CAN_00467]⌈If activation was successful then CanIf_CurrentIcomConfiguration shall be called with the parameter Error set to ICOM_SWITCH_E_OK. If activation was not successful then CanIf_CurrentIcomConfiguration shall be called with the parameter Error set to ICOM_SWITCH_E_FAILED. ⌋()

[SWS_CAN_00468]⌈If Pretended Networking is activated without having hardware support (software implementation – variant a) the CAN driver shall disable receive interrupts for messages, which are not relevant for wakeup (only messages which are configured as wakeup causes shall generate wakeup interrupts). ⌋()

[SWS_CAN_00469]⌈ If Pretended Networking is activated with CanIcomVariant set to CAN_ICOM_VARIANT_HW (implementation variant b + c in Figure 7-6) the CAN driver shall disable all receive interrupts. ⌋()

[SWS_CAN_00470]⌈If Pretended Networking is activated the CAN driver shall disable interrupts for tx messages. ⌋()

[SWS_CAN_00498]⌈ The CAN driver shall deactivate Pretended Networking before the CAN Controller is started by SetControllerMode(CAN_T_START)⌋()

Deactivation of Pretended Networking:

[SWS_CAN_00471]⌈Pretended Networking shall be deactivated by calling Can_SetIcomConfiguration with a configuration ID = 0. ⌋()

[SWS_CAN_00472]⌈If Pretended Networking is deactivated the CAN driver shall enable interrupts for all receive messages which are configured in the normal configuration. ⌋()

[SWS_CAN_00473]⌈If Pretended Networking is deactivated the CAN driver shall enable interrupts for tx messages. ⌋()

[SWS_CAN_00474]⌈ CAN driver shall inform CanIf about a configuration switch by calling CanIf_CurrentIcomConfiguration. The error parameter is set to ICOM_SWITCH_E_OK if deactivation is successful and to ICOM_SWITCH_E_FAILED otherwise.⌋()

[SWS_CAN_00475]⌈ If development error detection for the Can module is enabled, then function Can_SetIcomConfiguration shall report the development error CAN_E_ICOM_CONFIG_INVALID if it is called with an invalid ConfigurationId. ⌋()

[SWS_CAN_00499]⌈ The CAN driver shall deactivate Pretended Networking before the CAN Controller is stopped by SetControllerMode(CAN_T_STOP).⌋()

7.10.2 Support autonomous sending and receiving of messages

[SWS_CAN_00477]⌈Autonomous sending of messages in Pretended Networking mode shall be supported only if additional ICOM hardware is available. A configuration parameter defines if there is hardware support on not (Refer to CanIcomVariant). ⌋()

[SWS_CAN_00478]⌈If the ICOM is implemented in software it must not send messages in Pretended Networking mode. ⌋()

[SWS_CAN_00479]⌈ CanDriver shall forward all received messages received during Pretended Networking Mode to CanIf.⌋()

7.11 Error classification

[SWS_Can_00104] ⌈The Can module shall be able to detect the following errors and exceptions depending on its configuration

(development/production)」(SRS_BSW_00337, SRS_BSW_00385, SRS_BSW_00331)

Type or error	*Relevance*	*Related error code*	*Value [hex]*
API Service called with wrong parameter	Development	CAN_E_PARAM_POINTER CAN_E_PARAM_HANDLE CAN_E_PARAM_DLC CAN_E_PARAM_CONTROLLER	0x01 0x02 0x03 0x04
API Service used without initialization	Development	CAN_E_UNINIT	0x05
Invalid transition for the current mode	Development	CAN_E_TRANSITION	0x06
Received CAN message is lost	Development	CAN_E_DATALOST	0x07
Parameter Baudrate has an invalid value	Development	CAN_E_PARAM_BAUDRATE	0x08
Invalid ICOM Configuration Id	Development	CAN_E_ICOM_CONFIG_INVALID	0x09

7.11.1 Development Errors

[SWS_Can_00026] 「The Can module shall indicate errors that are caused by erroneous usage of the Can module API. This covers API parameter checks and call sequence errors. 」(SRS_BSW_00337, SRS_BSW_00323, SRS_SPAL_00157)

[SWS_Can_00091] 「After return of the DET the Can module's function that raised the development error shall return immediately.」(SRS_SPAL_12448)

[SWS_Can_00089] 「The Can module's environment shall indicate development errors only in the return values of a function of the Can module when DET is switched on and the function provides a return value. The returned value is CAN_NOT_OK. 」(SRS_BSW_00369, SRS_BSW_00386, SRS_SPAL_12448)

7.11.2 Production Errors

The Can module does not call the Diagnostic Event Manager, because there is no production error code defined for the Can module.

7.11.3 Return Values

CAN_BUSY is reported via return value of the function Can_Write. The CanIf module reacts according the sequence diagrams specified for the CanIf module.CAN_NOT_OK is reported via return value in case of a wakeup during transition to sleep mode.Bus-off and Wake-up events are forwarded via notification callback functions.

7.12 CAN FD Support

For performance reasons some CAN controllers allow to use a Flexible Data-Rate feature called CAN FD (see "CAN with Flexible Data-Rate" specification). Indicated during the arbitration phase it is possible to switch to a higher baud rate during payload and CRC. This second baud rate has to be configured by extending CanControllerBaudrateConfig with CanControllerFdBaudrateConfig. If a baud rate is active which has a CAN FD configuration (see CanControllerFdBaudrateConfig) the CAN FD feature is enabled for this controller. The specified second baud rate is needed to support reception of CAN FD frames with bit rate switch (BRS). Whether the second baudrate is used for transmission or not depends on configuration parameter CanControllerTxBitRateSwitch (see CanControllerFdBaudrateConfig).

However, there may be cases where conventional CAN 2.0 messages need to betransmitted in networks supporting CAN-FD messages for example to facilitate CAN selective wakeup. In these cases it is necessary to support transmitting interleaved conventional CAN messages with CAN-FD messages. This can be achieved on frame level by using the two most significant bits of the CanId (see Can_IdType, SWS_Can_00416) passed during Can_Write to indicate which kind of frame shall be used. Additionally, it is possible to switch to a non CAN FD baud rate using Can_SetBaudrate() which forces CanDrv to ignore the CAN FD flag in the CanId parameter.

8 API specification

The prefix of the function names may be changed in an implementation with several Can modules as described in SWS_Can_00284.

8.1 Imported types

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

[SWS_Can_00222] ⌈

Module	*Imported Type*
CanIf	CanIf_ControllerModeType
Can_GeneralTypes	Can_HwHandleType
	Can_ReturnType
	Can_StateTransitionType
	Can_HwType
	Can_PduType
ComStack_Types	IcomConfigIdType
	IcomSwitch_ErrorType
	PduIdType
	PduInfoType
Dem	Dem_EventIdType
	Dem_EventStatusType
EcuM	EcuM_WakeupSourceType
Icu	Icu_ChannelType
Os	CounterType
	StatusType
	TickRefType
Std_Types	Std_ReturnType
	Std_VersionInfoType

⌋()

8.2 Type definitions

[SWS_CAN_00487]⌈ The types specified in Can_GeneralTypes shall be declared in Can_GeneralTypes.h ⌋()

[SWS_Can_00439]⌈The content of Can_GeneralTypes.h shall be protected by a CAN_GENERAL_TYPES define. ⌋()

[SWS_Can_00440]⌈If different CAN drivers are used, only one instance of this file has to be included in the source tree. For implementation all Can_GeneralTypes.h related types in the documents mentioned before shall be considered.⌋()

8.2.1 Can_ConfigType

[SWS_Can_00413] ⌈

Name:	Can_ConfigType
Type:	Structure
Range:	Implementation specific.
Description:	This is the type of the external data structure containing the overall initialization data for the CAN driver and SFR settings affecting all controllers. Furthermore it contains pointers to controller configuration structures. The contents of the initialization data structure are CAN hardware specific.

⌋()

8.2.2 Can_PduType

[SWS_Can_00415] ⌈

Name:	Can_PduType		
Type:	Structure		
Element:	PduIdType	swPduHandle	--
	uint8	length	--
	Can_IdType	id	--
	uint8*	sdu	--
Description:	This type is used to provide ID, DLC and SDU from CAN interface to CAN driver.		

⌋()

8.2.3 Can_IdType

[SWS_Can_00416] ⌈

Name:	Can_IdType		
Type:	uint16, uint32		
Range:	Standard16Bit	--	0..0x47FF
	Standard32Bit	--	0..0x400007FF
	Extended32Bit	--	0..0xDFFFFFFF
Description:	Represents the Identifier of an L-PDU. The two most significant bits specify the frame type: 00 CAN message with Standard CAN ID 01 CAN FD frame with Standard CAN ID 10 CAN message with Extended CAN ID 11 CAN FD frame with Extended CAN ID		

⌋()

8.2.4 Can_HwHandleType

[SWS_Can_00429] ⌈

Name:	Can_HwHandleType

Type:	uint8, uint16		
Range:	Standard	--	0..0x0FF
	Extended	--	0..0xFFFF
Description:	Represents the hardware object handles of a CAN hardware unit. For CAN hardware units with more than 255 HW objects use extended range.		

⌋()

8.2.5 Can_HwType

[SWS_CAN_00496]⌈

Name:	Can_HwType		
Type:	Structure		
Element:	Can_IdType	CanId	Standard/Extended CAN ID of CAN L-PDU
	Can_HwHandleType	Hoh	ID of the corresponding Hardware Object Range
	uint8	ControllerId	ControllerId provided by CanIf clearly identify the corresponding controller
Description:	This type defines a data structure which clearly provides an Hardware Object Handle including its corresponding CAN Controller and therefore CanDrv as well as the specific CanId.		

⌋()

8.2.6 Can_StateTransitionType

[SWS_Can_00417] ⌈

Name:	Can_StateTransitionType	
Type:	Enumeration	
Range:	CAN_T_START	CAN controller transition value to request state STARTED.
	CAN_T_STOP	CAN controller transition value to request state STOPPED.
	CAN_T_SLEEP	CAN controller transition value to request state SLEEP.
	CAN_T_WAKEUP	CAN controller transition value to request state STOPPED from state SLEEP.
Description:	State transitions that are used by the function CAN_SetControllerMode	

⌋()

8.2.7 Can_ReturnType

[SWS_Can_00039] ⌈

Name:	Can_ReturnType	
Type:	Enumeration	
Range:	CAN_OK	success
	CAN_NOT_OK	error occurred or wakeup event occurred during sleep transition
	CAN_BUSY	transmit request could not be processed because no transmit object was available
Description:	Return values of CAN driver API .	

」(SRS_BSW_00331)

8.3 Function definitions

This is a list of functions provided for upper layer modules.

8.3.1 Services affecting the complete hardware unit

8.3.1.1 Can_Init

[SWS_Can_00223] 「

Service name:	Can_Init	

Syntax:	`void Can_Init(`
	`  const Can_ConfigType* Config`
	`)`	

Service ID[hex]:	0x00	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	Config	Pointer to driver configuration.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This function initializes the module.	

」(SRS_BSW_00358, SRS_BSW_00414)

Symbolic names of the available configuration sets are provided by the configuration description of the Can module. See chapter 10 about configuration description.

[SWS_Can_00174] 「If development error detection for the Can module is enabled: The function Can_Init shall raise the error CAN_E_TRANSITION if the driver is not in state CAN_UNINIT.」()

[SWS_Can_00408] 「If development error detection for the Can module is enabled: The function Can_Init shall raise the error CAN_E_TRANSITION if the CAN controllers are not in state UNINIT.」()

[SWS_Can_00175] 「If development error detection for the Can module is enabled: The function Can_Init shall raise the error CAN_E_PARAM_POINTER if a NULL pointer was given as config parameter.」()

8.3.1.2 Can_GetVersionInfo

[SWS_Can_00224] ⌈

Service name:	Can_GetVersionInfo	

Syntax:	`void Can_GetVersionInfo(`
	`  Std_VersionInfoType* versioninfo`
	`)`	

Service ID[hex]:	0x07	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	versioninfo	Pointer to where to store the version information of this module.
Return value:	None	
Description:	This function returns the version information of this module.	

⌋()

[SWS_Can_00177] ⌈If development error detection for the Can module is enabled: The function Can_GetVersionInfo shall raise the error CAN_E_PARAM_POINTER if the parameter versionInfo is a null pointer.⌋()

8.3.1.3 Can_CheckBaudrate

[SWS_Can_00454] ⌈

Service name:	Can_CheckBaudrate	

Syntax:	`Std_ReturnType Can_CheckBaudrate(`
	`  uint8 Controller,`
	`  const uint16 Baudrate`
	`)`	

Service ID[hex]:	0x0e	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	Controller	CAN Controller to check for the support of a certain baudrate
	Baudrate	Baudrate to check in kbps
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Baudrate supported by the CAN Controller E_NOT_OK: Baudrate not supported / invalid CAN controller
Description:	This service shall check, if a certain CAN controller supports a requested baudrate Please note that this API is deprecated and is kept only for backward compatibility reasons. In the next major release this API will be deleted.	

」()

[SWS_Can_00455] 「The service `Can_CheckBaudrate(Controller, Baudrate)` shall be called by `CanIf_CheckBaudrate()` for the requested CAN controller. 」()

[SWS_Can_00456] 「If the CAN Driver module was not initialized before calling `Can_CheckBaudrate(Controller, Baudrate)` and if development error detection is enabled (i.e. `CAN_DEV_ERROR_DETECT` equals `ON`), then the Can shall report development error code `CAN_E_UNINIT` to the `Det_ReportError` service of the DET module.」()

[SWS_Can_00457] 「If parameter `Controller` of `Can_CheckBaudrate(Controller, Baudrate)` has an invalid value and if development error detection is enabled (i.e. `CAN_DEV_ERROR_DETECT` equals `ON`), then the Can shall report development error code `CAN_E_PARAM_CONTROLLER` to the `Det_ReportError` service of the DET module.」()

[SWS_Can_00458] 「If parameter `Baudrate` of `Can_CheckBaudrate(Controller, Baudrate)` has an invalid value and if development error detection is enabled (i.e. `CAN_DEV_ERROR_DETECT` equals `ON`), then the Can shall report development error code `CAN_E_PARAM_BAUDRATE` to the `Det_ReportError` service of the DET module.」()

[SWS_Can_00459] 「Caveats of `Can_CheckBaudrate(Controller, Baudrate)`:

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

[SWS_Can_00460] 「Configuration of Can_CheckBaudrate(Controller, Baudrate): If Can supports changing of the baudrate and thus this service, shall be configurable via CAN_CHANGE_BAUDRATE_API」()

8.3.2 Services affecting one single CAN Controller

8.3.2.1 Can_ChangeBaudrate

[SWS_Can_00449] 「

Service name:	Can_ChangeBaudrate	
Syntax:	`Std_ReturnType Can_ChangeBaudrate( uint8 Controller, const uint16 Baudrate )`	
Service ID[hex]:	0x0d	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	Controller	CAN Controller, whose baudrate shall be changed
	Baudrate	Requested baudrate in kbps

Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Service request accepted, baudrate change started E_NOT_OK: Service request not accepted
Description:	This service shall change the baudrate of the CAN controller. Please note that this API is deprecated and is kept only for backward compatibility reasons. Can_SetBaudrate API shall be used instead to change the baud rate configuration. In the next major release this API will be deleted.	

⌋()

The function Can_ChangeBaudrate re-initializes the CAN controller and the controller specific settings (see SWS_Can_00062).
Different sets of static configuration may have been configured. The parameter *Config points to the hardware specific structure that describes the configuration (see SWS_Can_00291).
Global CAN Hardware Unit settings must not be changed. Only a subset of parameters may be changed during runtime (see chapter 10). For further explanation, see also chapter 7.4
The CAN controller must be in state STOPPED when this function is called (see SWS_Can_00256 and SWS_Can_00260).
The CAN controller is in state STOPPED after (re-)initialization (see SWS_Can_00259).

[SWS_Can_00450] ⌈If development error detection for the Can module is enabled: The function `Can_ChangeBaudrate` shall raise the error `CAN_E_UNINIT` if the driver is not yet initialized.⌋()

[SWS_Can_00451] ⌈If development error detection for the Can module is enabled: The function `Can_ChangeBaudrate` shall raise the error `CAN_E_PARAM_BAUDRATE` if the parameter Baudrate has an invalid value.⌋()

[SWS_Can_00452] ⌈If development error detection for the Can module is enabled: The function `Can_ChangeBaudrate` shall raise the error `CAN_E_PARAM_CONTROLLER` if the parameter `Controller` is out of range.⌋()

[SWS_Can_00453] ⌈If development error detection for the Can module is enabled: if the controller is not in state STOPPED, the function Can_ChangeBaudrate shall raise the error CAN_E_TRANSITION.⌋()

[SWS_Can_00461] ⌈If hardware supports wake-up (i.e. CanWakeupSupport == true), it shall be checked during controller initialization if there was a wake-up event on the specific CAN controller. If a wake-up event has been detected, the wake-up shall directly be reported to the EcuM via EcuM_SetWakeupEvent call-back function.⌋()

8.3.2.2 Can_SetBaudrate

[SWS_CAN_00491]⌜

Service name:	Can_SetBaudrate	
Syntax:	`Std_ReturnType Can_SetBaudrate( uint8 Controller, uint16 BaudRateConfigID )`	
Service ID[hex]:	0x0f	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant for different Controllers. Non reentrant for the same Controller.	
Parameters (in):	Controller	CAN controller, whose baud rate shall be set
	BaudRateConfigID	references a baud rate configuration by ID (see CanControllerBaudRateConfigID)
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Service request accepted, setting of (new) baud rate started E_NOT_OK: Service request not accepted
Description:	This service shall set the baud rate configuration of the CAN controller. Depending on necessary baud rate modifications the controller might have to reset.	

⌟()

There might be several baud rate configurations available. The function Can_SetBaudrate can be used to switch between different configurations. Depending on the old and new baud rate configuration only a subset of parameters may be changed during runtime and a re-initialization of the CAN Controller might be avoidable.

If the call of Can_SetBaudrate will cause a re-initialization of the CAN Controller the CAN controller must be in state STOPPED when this function is called (see SWS_Can_00256 and SWS_Can_00260).
The CAN controller is in state STOPPED after (re-)initialization (see SWS_Can_00259).

[SWS_CAN_00492]⌜ If development error detection for the Can module is enabled: The function Can_SetBaudrate shall raise the error CAN_E_UNINIT and return E_NOT_OK if the driver is not yet initialized.⌟()

[SWS_CAN_00493]⌜ If development error detection for the Can module is enabled: The function Can_SetBaudrate shall raise the error CAN_E_PARAM_BAUDRATE and return E_NOT_OK if the parameter BaudRateConfigID has an invalid value.⌟()

[SWS_CAN_00494]⌜ If development error detection for the Can module is enabled the function Can_SetBaudrate shall raise the error CAN_E_PARAM_CONTROLLER and return E_NOT_OK if the parameter Controller is out of range.⌟()

8.3.2.3 Can_SetControllerMode

[SWS_Can_00230] ⌈

Service name:	Can_SetControllerMode	
Syntax:	`Can_ReturnType Can_SetControllerMode( uint8 Controller, Can_StateTransitionType Transition )`	
Service ID[hex]:	0x03	
Sync/Async:	Asynchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	Controller	CAN controller for which the status shall be changed
	Transition	Transition value to request new CAN controller state
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Can_ReturnType	CAN_OK: request accepted CAN_NOT_OK: request not accepted, a development error occurred
Description:	This function performs software triggered state transitions of the CAN controller State machine.	

⌋()

[SWS_Can_00017] ⌈The function Can_SetControllerMode shall perform software triggered state transitions of the CAN controller State machine. See also [SRS_SPAL_12169]⌋(SRS_SPAL_12169, SRS_Can_01053)

[SWS_Can_00384] ⌈Each time the CAN controller state machine is triggered with the state transition value CAN_T_START, the function Can_SetControllerMode shall re-initialize the CAN controller with the same controller configuration set previously used by functions Can_SetBaudrate or Can_Init.⌋()
Refer to SWS_Can_00048 for the case of a wakeup event from CAN bus occurred during sleep transition.

[SWS_Can_00294] ⌈The function Can_SetControllerMode shall disable the wake-up interrupt, while checking the wake-up status. ⌋()

[SWS_Can_00196] ⌈The function Can_SetControllerMode shall enable interrupts that are needed in the new state. ⌋()

[SWS_Can_00425] ⌈Enabling of CAN interrupts shall not be executed, when CAN interrupts have been disabled by function Can_DisableControllerInterrupts.⌋()

[SWS_Can_00197] ⌈The function Can_SetControllerMode shall disable interrupts that are not allowed in the new state. ⌋()

[SWS_Can_00426] ⌈Disabling of CAN interrupts shall not be executed, when CAN interrupts have been disabled by function Can_DisableControllerInterrupts.⌋()
Caveat:

The behavior of the transmit operation is undefined when the ‘software’ state in the CanIf module is already CANIF_CS_STARTED, but the CAN controller is not yet in operational mode.
The CanIf module must ensure that the function is not called before the previous call of Can_SetControllerMode returned.
The CanIf module is responsible not to initiate invalid transitions.

[SWS_Can_00198] ⌈If development error detection for the Can module is enabled: if the module is not yet initialized, the function Can_SetControllerMode shall raise development error CAN_E_UNINIT and return CAN_NOT_OK.⌋()

[SWS_Can_00199] ⌈If development error detection for the Can module is enabled: if the parameter `Controller` is out of range, the function Can_SetControllerMode shall raise development error CAN_E_PARAM_CONTROLLER and return CAN_NOT_OK.⌋()

[SWS_Can_00200] ⌈If development error detection for the Can module is enabled: if an invalid transition has been requested, the function Can_SetControllerMode shall raise the error CAN_E_TRANSITION and return CAN_NOT_OK.⌋()

8.3.2.4 Can_DisableControllerInterrupts

[SWS_Can_00231] ⌈

Service name:	Can_DisableControllerInterrupts	
Syntax:	`void Can_DisableControllerInterrupts( uint8 Controller )`	
Service ID[hex]:	0x04	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	Controller	CAN controller for which interrupts shall be disabled.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This function disables all interrupts for this CAN controller.	

⌋(SRS_BSW_00312)

[SWS_Can_00049] ⌈ The function Can_DisableControllerInterrupts shall access the CAN controller registers to disable all interrupts for that CAN controller only, if interrupts for that CAN Controller are enabled.⌋ (SRS_Can_01043)

[SWS_Can_00202] ⌈When Can_DisableControllerInterrupts has been called several times, Can_EnableControllerInterrupts must be called as many times before the interrupts are re-enabled.⌋()

Implementation note:
The function Can_DisableControllerInterrupts can increase a counter on every execution that indicates how many Can_EnableControllerInterrupts need to be called before the interrupts will be enabled (incremental disable).

[SWS_Can_00204] ⌈The Can module shall track all individual enabling and disabling of interrupts in other functions (i.e. Can_SetControllerMode) , so that the correct interrupt enable state can be restored.⌋()

Implementation example:
- in 'interrupts enabled mode': For each interrupt state change does not only modify the interrupt enable bit, but also a software flag.
- in 'interrupts disabled mode': only the software flag is modified.
- Can_DisableControllerInterrupts and Can_EnableControllerInterrupts do not modify the software flags.
- Can_EnableControllerInterrupts reads the software flags to re-enable the correct interrupts.

[SWS_Can_00205] ⌈If development error detection for the Can module is enabled: The function Can_DisableControllerInterrupts shall raise the error CAN_E_UNINIT if the driver not yet initialized.⌋()

[SWS_Can_00206] ⌈If development error detection for the Can module is enabled: The function Can_DisableControllerInterrupts shall raise the error CAN_E_PARAM_CONTROLLER if the parameter `Controller` is out of range.⌋()

8.3.2.5 Can_EnableControllerInterrupts

[SWS_Can_00232] ⌈

Service name:	Can_EnableControllerInterrupts	

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

Service ID[hex]:	0x05	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	Controller	CAN controller for which interrupts shall be re-enabled
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This function enables all allowed interrupts.	

」(SRS_BSW_00312)

[SWS_Can_00050] 「The function Can_EnableControllerInterrupts shall enable all interrupts that must be enabled according the current software status.」(SRS_Can_01043)
SWS_Can_00202 applies to this function.
[SWS_Can_00208] 「The function Can_EnableControllerInterrupts shall perform no action when Can_DisableControllerInterrupts has not been called before.」()
See also implementation example for Can_DisableControllerInterrupts.
[SWS_Can_00209] 「If development error detection for the Can module is enabled: The function Can_EnableControllerInterrupts shall raise the error CAN_E_UNINIT if the driver not yet initialized.」()

[SWS_Can_00210] 「If development error detection for the Can module is enabled: The function Can_EnableControllerInterrupts shall raise the error CAN_E_PARAM_CONTROLLER if the parameter `Controller` is out of range.」()

8.3.2.6 Can_CheckWakeup

[SWS_Can_00360] 「

Service name:	Can_CheckWakeup	
Syntax:	`Can_ReturnType Can_CheckWakeup( uint8 Controller )`	
Service ID[hex]:	0x0b	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	Controller	Controller to be checked for a wakeup.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Can_ReturnType	CAN_OK: A wakeup was detected for the given controller. CAN_NOT_OK: No wakeup was detected for the given controller.
Description:	This function checks if a wakeup has occurred for the given controller.	

」()

[SWS_Can_00361] 「The function Can_CheckWakeup shall check if the requested CAN controller has detected a wakeup. If a wakeup event was successfully detected since the last go to SLEEP, the function shall return CAN_OK, otherwise CAN_NOT_OK.」()

[SWS_CAN_00484] ⌈The function Can_CheckWakeup shall check if a wakeup capable CAN controller is the source for a wakeup event and call EcuM_SetWakeupEvent to indicate the ECU State Manager. ⌋()

[SWS_CAN_00485]⌈ The function Can_CheckWakeup shall be pre compile time configurable On/Off by the configuration parameter: CanWakeupFunctionalityAPI ⌋()

[SWS_Can_00362] ⌈If development error detection for the Can module is enabled: The function Can_CheckWakeup shall raise the error CAN_E_UNINIT if the driver is not yet initialized.⌋()

[SWS_Can_00363] ⌈If development error detection for the Can module is enabled: The function Can_CheckWakeup shall raise the error CAN_E_PARAM_CONTROLLER if the parameter `Controller` is out of range.⌋()

8.3.3 Services affecting a Hardware Handle

8.3.3.1 Can_Write

[SWS_Can_00233] ⌈

Service name:	Can_Write	

Syntax:	`Can_ReturnType Can_Write(`
	`  Can_HwHandleType Hth,`
	`  const Can_PduType* PduInfo`
	`)`	

Service ID[hex]:	0x06	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant (thread-safe)	
Parameters (in):	Hth	information which HW-transmit handle shall be used for transmit. Implicitly this is also the information about the controller to use because the Hth numbers are unique inside one hardware unit.
	PduInfo	Pointer to SDU user memory, DLC and Identifier.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Can_ReturnType	CAN_OK: Write command has been accepted CAN_NOT_OK: development error occurred CAN_BUSY: No TX hardware buffer available or pre-emptive call of Can_Write that can't be implemented re-entrant
Description:	This function is called by CanIf to pass a CAN message to CanDrv for transmission.	

⌋(SRS_BSW_00312)

The function Can_Write first checks if the hardware transmit object that is identified by the HTH is free and if another Can_Write is ongoing for the same HTH.

[SWS_Can_00212] ⌈The function Can_Write shall perform following actions if the hardware transmit object is free:

- The mutex for that HTH is set to 'signaled'
- the ID, DLC and SDU are put in a format appropriate for the hardware (if necessary) and copied in the appropriate hardware registers/buffers.
- All necessary control operations to initiate the transmit are done
- The mutex for that HTH is released
- The function returns with CAN_OK⌋(SRS_Can_01049)

[SWS_Can_00213] ⌈The function Can_Write shall perform no actions if the hardware transmit object is busy with another transmit request for an L-PDU that has **higher** priority than that for the current request:

1. The transmission of the L-PDU with higher priority shall not be cancelled and the function Can_Write is left without any actions.
2. The function Can_Write shall return CAN_BUSY⌋(SRS_Can_01049)

[SWS_Can_00215] ⌈The function Can_Write shall perform following actions if the hardware transmit object is busy with another transmit request for an L-PDU that has **lower** priority than that for the current request:

3. The transmission of the L-PDU with lower priority shall be cancelled (asynchronously) in case transmit cancellation functionality is enabled. Compare to chapter 7.5.1.2.
4. The function Can_Write shall return CAN_BUSY⌋()

[SWS_Can_00434] ⌈The function Can_Write shall perform following actions if the hardware transmit object is busy with another transmit request for an L-PDU that has **identical** priority than that for the current request:

5. The transmission of the L-PDU with identical priority shall be cancelled (asynchronously) in case *CanIdenticalIdCancellation* is enabled. Compare to chapter 7.5.1.2.
6. The transmission of the L-PDU with identical priority shall not be cancelled in case *CanIdenticalIdCancellation* is disabled and the function Can_Write is left without any actions.
7. The function Can_Write shall return CAN_BUSY⌋()

[SWS_Can_00214] ⌈The function Can_Write shall return CAN_BUSY if a preemptive call of Can_Write has been issued, that could not be handled reentrant (i.e. a call with the same HTH).⌋(SRS_BSW_00312, SRS_Can_01049)

[SWS_Can_00275] ⌈The function Can_Write shall be non-blocking.⌋()

[SWS_Can_00216] ⌈If development error detection for the Can module is enabled: The function Can_Write shall raise the error CAN_E_UNINIT and shall return CAN_NOT_OK if the driver is not yet initialized.⌋()

[SWS_Can_00217] ⌈If development error detection for the Can module is enabled: The function Can_Write shall raise the error CAN_E_PARAM_HANDLE and shall return CAN_NOT_OK if the parameter `Hth` is not a configured Hardware Transmit Handle.⌋()

[SWS_Can_00218] ⌈If development error detection for the Can module is enabled: The function Can_Write shall raise the error CAN_E_PARAM_DLC and shall return CAN_NOT_OK if the length is more than 8 byte.⌋()

[SWS_Can_00219] ⌈If development error detection for the Can module is enabled: The function Can_Write shall raise the error CAN_E_PARAM_POINTER and shall return CAN_NOT_OK if the parameter `PduInfo` or the SDU pointer inside PduInfo is a null-pointer.⌋()

[SWS_CAN_00486]⌈ The CAN Frame has to be sent according to the two most significant bits of Can_PduType->id. The CAN FD frame bit is only evaluated if FD baud rate is active.⌋()

8.4 Call-back notifications

This chapter lists all functions provided by the Can module to lower layer modules. The lower layer module of Can module is the SPI module. The SPI module, which is part of the MCAL, may used to exchange data between the microcontroller and an external CAN controller.
The Can module does not provide callback functions. Only synchronous MCAL API may used to access external CAN controllers.

8.4.1 Call-out function

The AUTOSAR CAN module supports optional L-PDU callouts on every reception of a L-PDU.

[SWS_Can_00443] ⌈The L-PDU-Callout API shall be defined as:

```
FUNC(boolean, COM_APPL_CODE) <LPDU_CalloutName>
(
uint8         Hrh,
Can_IdType    CanId,
uint8         CanDlc,
const uint8   *CanSduPtr
)
```

⌋()

where <LPDU_CalloutName> has to be substituted with the concrete L-PDU callout name which is configurable, see SWS_Can_00434_Conf.

[SWS_Can_00444] ⌈If the L-PDU callout returns false, the L-PDU shall not be processed any further. ⌋()

8.4.2 Enabling/Disabling wakeup notification

[SWS_Can_00445] ⌈Can driver shall use the following APIs provided by Icu driver, to enable and disable the wakeup event notification:

- Icu_EnableNotification
- `Icu_DisableNotification`⌋()

[SWS_Can_00446] ⌈Icu_EnableNotification shall be called when “external” Can controllers have been transitioned to SLEEP state (CANIF_CS_SLEEP).⌋()

[SWS_Can_00447] ⌈Icu_DisableNotification “external” Can controllers have been transitioned to STOPPED state (CANIF_CS_STOPPED).⌋()

8.5 Scheduled functions

These functions are directly called by Basic Software Scheduler. The following functions shall have no return value and no parameter. All functions shall be non-reentrant.

[SWS_Can_00110] ⌈There is no requirement regarding the execution order of the CAN main processing functions.⌋(SRS_BSW_00428)

8.5.1.1 Can_MainFunction_Write

[SWS_Can_00225] ⌈

Service name:	Can_MainFunction_Write

Syntax:	`void Can_MainFunction_Write(`
	`  void`
	`)`

Service ID[hex]:	0x01
Description:	This function performs the polling of TX confirmation and TX cancellation confirmation when CAN_TX_PROCESSING is set to POLLING.

⌋()

[SWS_Can_00031] ⌈The function Can_MainFunction_Write shall perform the polling of TX confirmation and TX cancellation confirmation when CanTxProcessing is set to POLLING.⌋(SRS_BSW_00432, SRS_BSW_00373, SRS_BSW_00376, SRS_SPAL_00157)

[SWS_Can_00178] ⌈The Can module may implement the function Can_MainFunction_Write as empty define in case no polling at all is used.⌋()

[SWS_Can_00179] ⌈If development error detection for the module Can is enabled: The function Can_MainFunction_Write shall raise the error CAN_E_UNINIT if the driver is not yet initialized.⌋()

[SWS_Can_00441] ⌈The API name of Can_MainFunction_Write() shall obey the following pattern:

- Can_MainFunction_Wrtte_0()
- Can_MainFunction_Write_1()
- Can_MainFunction_Write_2()
- Can_MainFunction_Write_3()
- ... and so on, if more than one period (see ECUC_Can_00356) is supported.⌋()

8.5.1.2 Can_MainFunction_Read

[SWS_Can_00226] ⌈

Service name:	Can_MainFunction_Read

Syntax:	`void Can_MainFunction_Read(`
	`  void`
	`)`

Service ID[hex]:	0x08
Description:	This function performs the polling of RX indications when CAN_RX_PROCESSING is set to POLLING.

⌋()

[SWS_Can_00108] ⌈The function Can_MainFunction_Read shall perform the polling of RX indications when CanRxProcessing is set to POLLING.⌋(SRS_BSW_00432, SRS_SPAL_00157)

[SWS_Can_00180] ⌈The Can module may implement the function Can_MainFunction_Read as empty define in case no polling at all is used.⌋()

[SWS_Can_00181] ⌈If development error detection for the Can module is enabled: The function Can_MainFunction_Read shall raise the error CAN_E_UNINIT if the driver is not yet initialized.⌋()

[SWS_Can_00442] ⌈The API name of Can_MainFunction_Read() shall obey the following pattern:

- Can_MainFunction_Read_0()
- Can_MainFunction_Read_1()
- Can_MainFunction_Read_2()
- Can_MainFunction_Read_3()

• ... and so on, if more than one period (see ECUC_Can_00358) is supported.」()

8.5.1.3 Can_MainFunction_BusOff

[SWS_Can_00227] 「

Service name:	Can_MainFunction_BusOff

Syntax:	`void Can_MainFunction_BusOff(`
	`  void`
	`)`

Service ID[hex]:	0x09
Description:	This function performs the polling of bus-off events that are configured statically as 'to be polled'.

」()

[SWS_Can_00109] 「The function Can_MainFunction_BusOff shall perform the polling of bus-off events that are configured statically as 'to be polled'.」() (SRS_BSW_00432, SRS_SPAL_00157)

[SWS_Can_00183] 「The Can module may implement the function Can_MainFunction_BusOff as empty define in case no polling at all is used.」()

[SWS_Can_00184] 「If development error detection for the Can module is enabled: The function Can_MainFunction_BusOff shall raise the error CAN_E_UNINIT if the driver is not yet initialized.」()

8.5.1.4 Can_MainFunction_Wakeup

[SWS_Can_00228] 「

Service name:	Can_MainFunction_Wakeup

Syntax:	`void Can_MainFunction_Wakeup(`
	`  void`
	`)`

Service ID[hex]:	0x0a
Description:	This function performs the polling of wake-up events that are configured statically as 'to be polled'.

」()

[SWS_Can_00112] 「The function Can_MainFunction_Wakeup shall perform the polling of wake-up events that are configured statically as 'to be polled'.」(SRS_BSW_00432, SRS_SPAL_00157)

[SWS_Can_00185] ⌈The Can module may implement the function Can_MainFunction_Wakeup as empty define in case no polling at all is used.⌋()

[SWS_Can_00186] ⌈If development error detection for the Can module is enabled: The function Can_MainFunction_Wakeup shall raise the error CAN_E_UNINIT if the driver is not yet initialized.⌋()

8.5.1.5 Can_MainFunction_Mode

[SWS_Can_00368] ⌈

Service name:	Can_MainFunction_Mode

Syntax:	`void Can_MainFunction_Mode(`
	`  void`
	`)`

Service ID[hex]:	0x0c
Description:	This function performs the polling of CAN controller mode transitions.

⌋()

[SWS_Can_00369] ⌈The function Can_MainFunction_Mode shall implement the polling of CAN status register flags to detect transition of CAN Controller state. Compare to chapter 7.3.2.⌋()

[SWS_Can_00379] ⌈If development error detection for the Can module is enabled: The function Can_MainFunction_Mode shall raise the error CAN_E_UNINIT if the driver is not yet initialized.⌋()

8.6 Expected Interfaces

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

8.6.1 Mandatory Interfaces

This chapter defines all interfaces which are required to fulfill the core functionality of the module. All callback functions that are called by the Can module are implemented in the CanIf module. These callback functions are not configurable.

[SWS_Can_00234] ⌈

API function	***Description***
CanIf_ControllerBusOff	This service indicates a Controller BusOff event referring to the corresponding CAN Controller.
CanIf_ControllerModeIndication	This service indicates a controller state transition referring to the corresponding CAN controller.
CanIf_RxIndication	This service indicates a successful reception of a received CAN Rx L-

	PDU to the CanIf after passing all filters and validation checks.
CanIf_TxConfirmation	This service confirms a previously successfully processed transmission of a CAN TxPDU.
GetCounterValue	This service reads the current count value of a counter (returning either the hardware timer ticks if counter is driven by hardware or the software ticks when user drives counter).

⌋(SRS_BSW_00387, SRS_Can_01055)

8.6.2 Optional Interfaces

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

[SWS_Can_00235] ⌈

API function	*Description*
CanIf_CancelTxConfirmation	This service informs CanIf that a L-PDU shall be buffered in CanIf TxBuffer from CAN hardware object to avoid priority inversion.
CanIf_CurrentIcomConfiguration	This service shall inform about the change of the Icom Configuration of a CAN controller.
Dem_ReportErrorStatus	Queues the reported events from the BSW modules (API is only used by BSW modules). The interface has an asynchronous behavior, because the processing of the event is done within the Dem main function. OBD Events Suppression shall be ignored for this computation.
Det_ReportError	Service to report development errors.
EcuM_CheckWakeup	This callout is called by the EcuM to poll a wakeup source. It shall also be called by the ISR of a wakeup source to set up the PLL and check other wakeup sources that may be connected to the same interrupt.
EcuM_SetWakeupEvent	Sets the wakeup event.
Icu_DisableNotification	This function disables the notification of a channel.
Icu_EnableNotification	This function enables the notification on the given channel.

⌋(SRS_SPAL_12056, SRS_Can_01054)

8.6.3 Configurable interfaces

There is no configurable target for the Can module. The Can module always reports to CanIf module.

8.7 API supporting Pretended Networking

8.7.1.1 Can_SetIcomConfiguration

Service name:	Can_SetIcomConfiguration
Syntax:	`Std_ReturnType Can_SetIcomConfiguration(`

	`uint8 Controller,`
	`IcomConfigIdType ConfigurationId`
	)	

Service ID[hex]:	0xf	
Sync/Async:	Asynchronous	
Reentrancy:	Reentrant only for different controller Ids	
Parameters (in):	Controller	CAN controller for which the status shall be changed.
	ConfigurationId	Requested Configuration
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: CAN driver succeeded in setting a configuration with a valid Configuration id. E_NOT_OK: CAN driver failed to set a configuration with a valid Configuration id.
Description:	This service shall change the Icom Configuration of a CAN controller to the requested one.	

[SWS_CAN_00480]⌈ The interface Can_SetIcomConfiguration shall activate Pretended Networking and load the requested ICOM configuration for a given controller.⌋()

[SWS_CAN_00481]⌈ The function Can_SetIcomConfiguration shall reconfigure the controller with the ICOM configuration parameters specified by the ConfigurationId.⌋()

[SWS_CAN_00482]⌈ The rx and tx interrupts for message objects must be enabled or disabled, if this is supported by the controller.⌋()

[SWS_CAN_00483]⌈ If the ICOM is implemented in hardware, the ICOM hardware must be activated or deactivated.⌋**()**

[SWS_CAN_00495]⌈Can_SetIcomConfiguration() shall be pre compile time configurable ON/OFF by the configuration parameter CAN_PUBLIC_ICOM_SUPPORT.⌋()

9 Sequence diagrams

9.1 Interaction between Can and CanIf module

For sequence diagrams see the CanIf module Specification [5].
There are described the sequences for Transmission, Reception and Error Handling.

9.2 Wakeup sequence

For Wakeup sequence diagrams refer to Specification of ECU State Manager [7].

10 Configuration specification

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 Can module.

Chapter 10.3 specifies published information of the Can module.

10.1 How to read this chapter

For details refer to the chapter 10.1 “Introduction to configuration specification” in *SWS_BSWGeneral*

10.2 Containers and configuration parameters

The following chapters summarize all configuration parameters. The detailed meanings of the parameters describe Chapters 7 and Chapter 8.
The described parameters are input for the Can module configurator.

[SWS_Can_00022] ⌈The code configuration of the Can module is CAN controller specific. If the CAN controller is sited on-chip, the code generation tool for the Can module is µController specific. If the CAN controller is an external device, the generation tool must not be µController specific.⌋(SRS_BSW_00159)

[SWS_Can_00024] ⌈The valid values that can be configured are hardware dependent. Therefore the rules and constraints can’t be given in the standard. The configuration tool is responsible to do a static configuration checking, also regarding dependencies between modules (i.e. Port driver, MCU driver etc.)⌋(SRS_BSW_00167, SRS_SPAL_12463)

10.2.1 Variants

The Can module provides two variants of configuration sets:

[SWS_Can_00220] ⌈VARIANT-PRE-COMPILE: Only pre-compile configuration parameters.⌋()

[SWS_Can_00221] ⌈VARIANT-POST-BUILD: Mix of pre compile- and post build time configuration parameters.⌋()

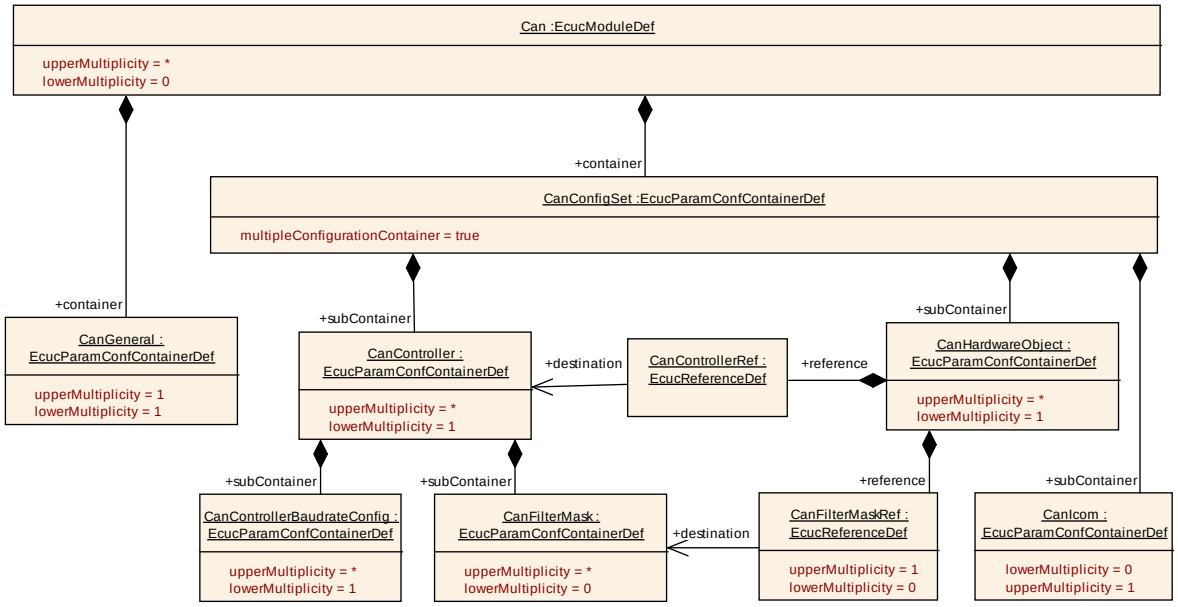


Figure 10-1: Can Module Configuration Layout

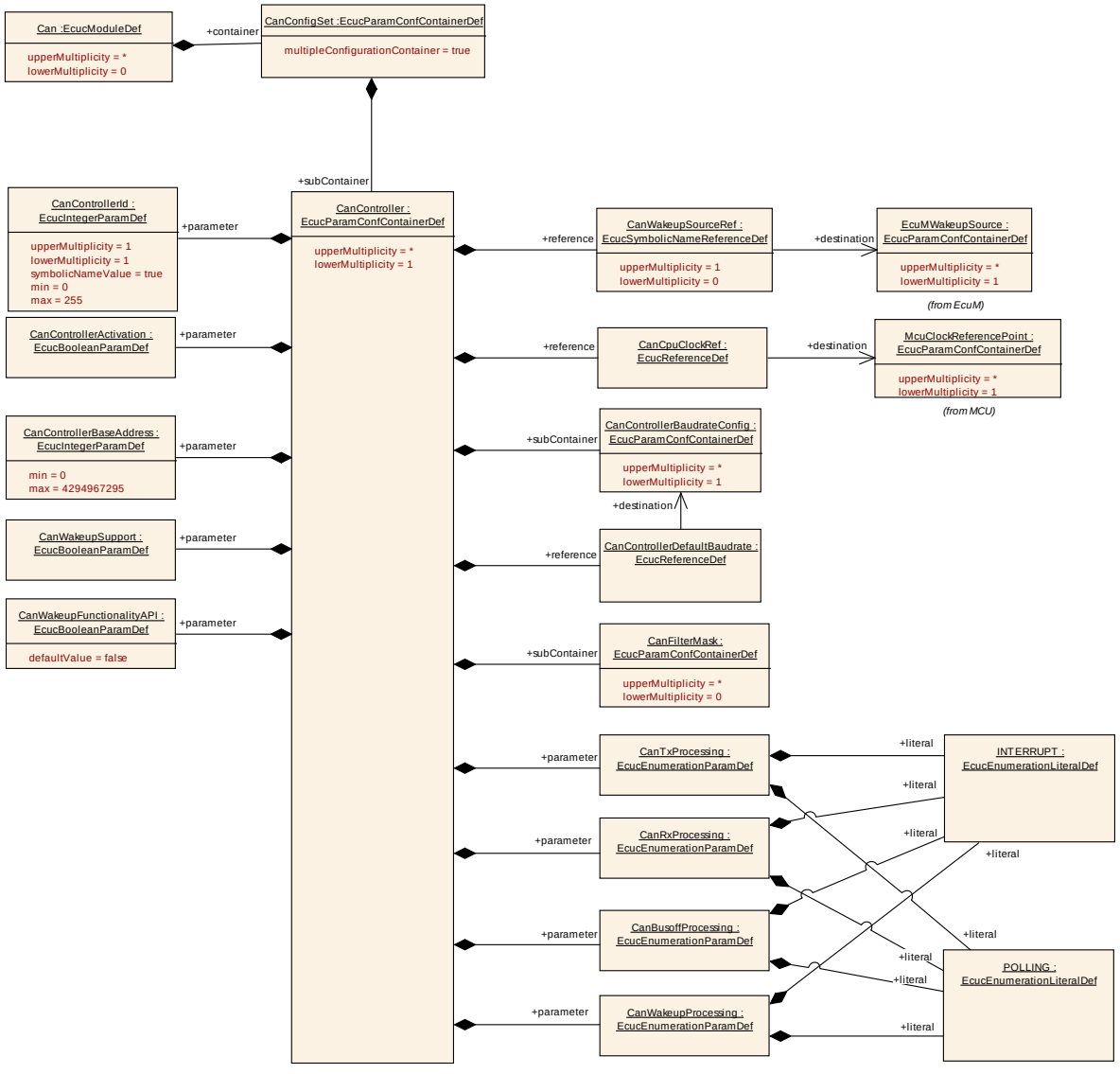


Figure 10-2: Can Controller Configuration Layout

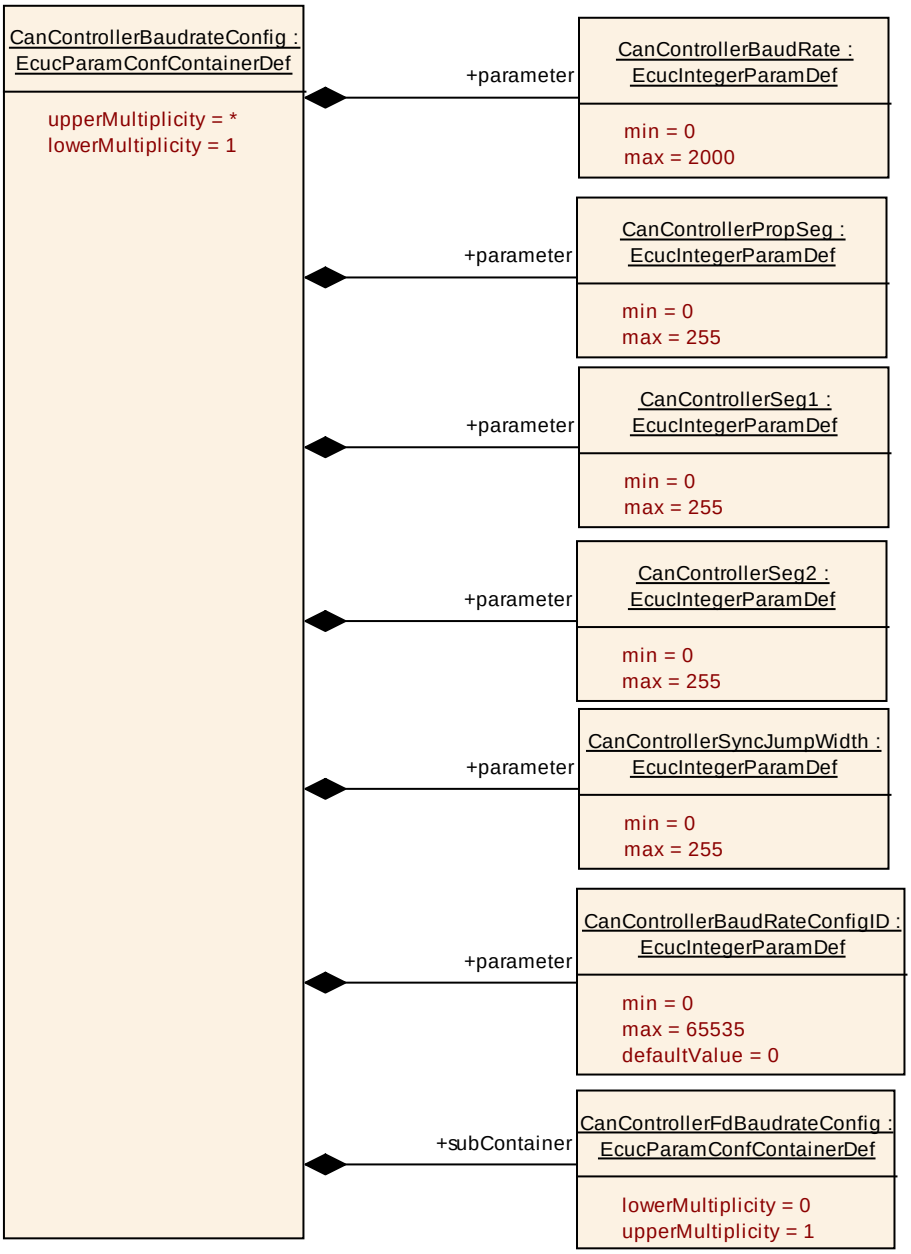


Figure 10-3: Can Controller Baud Rate Configuration Layout

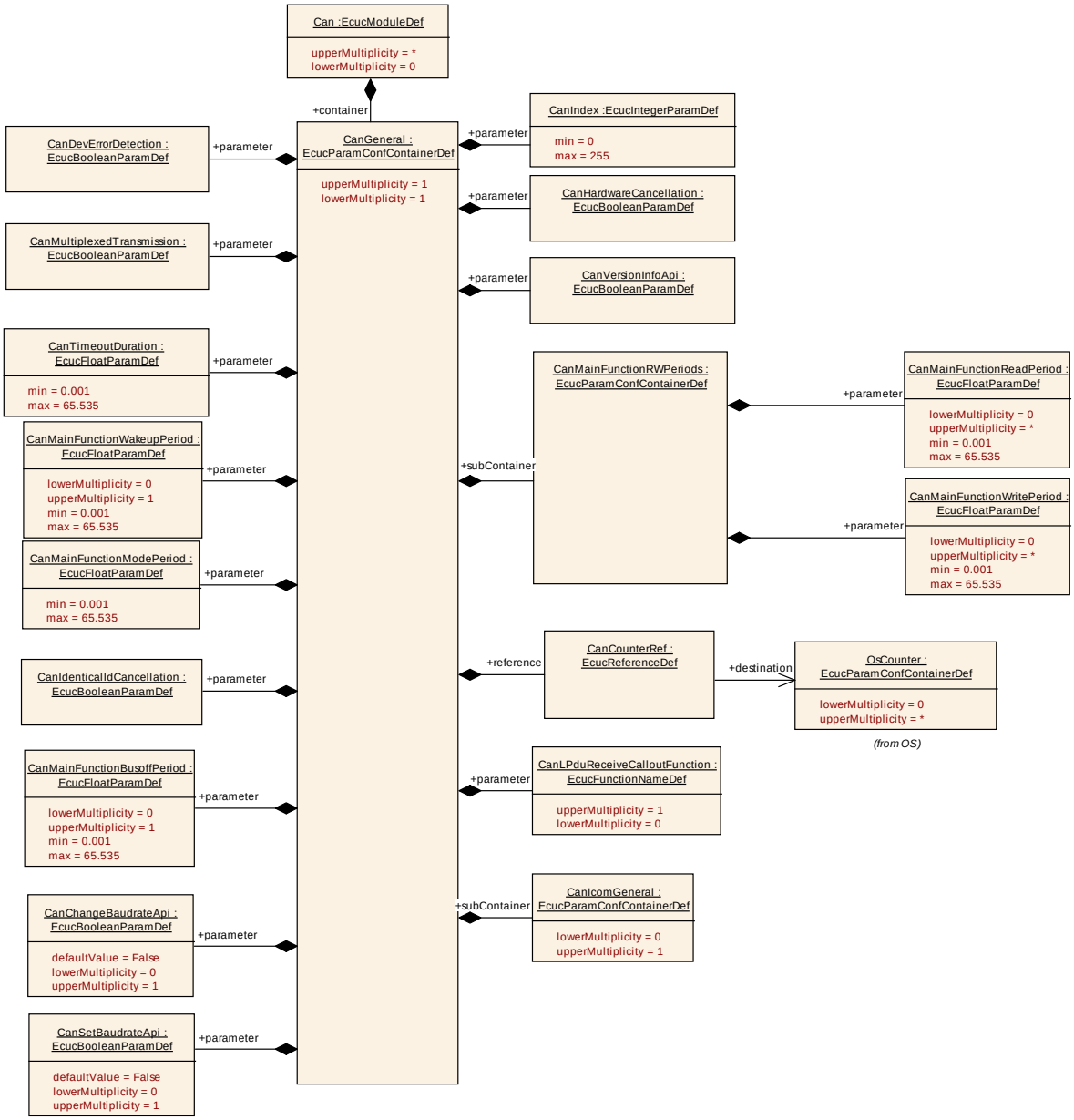


Figure 10-4: Can General Configuration Layout

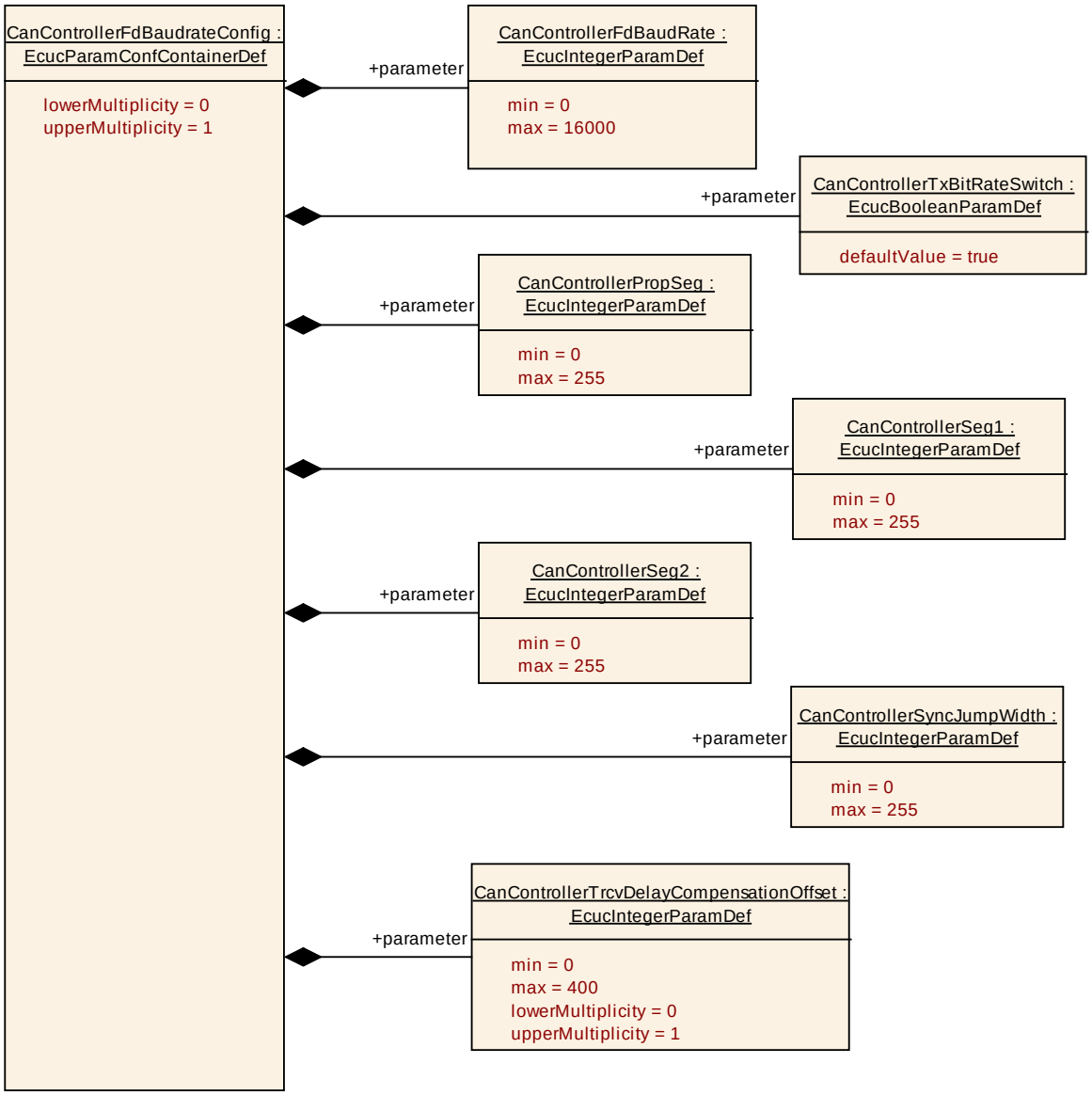


Figure 10-5: CanControllerFdBaudrateConfig

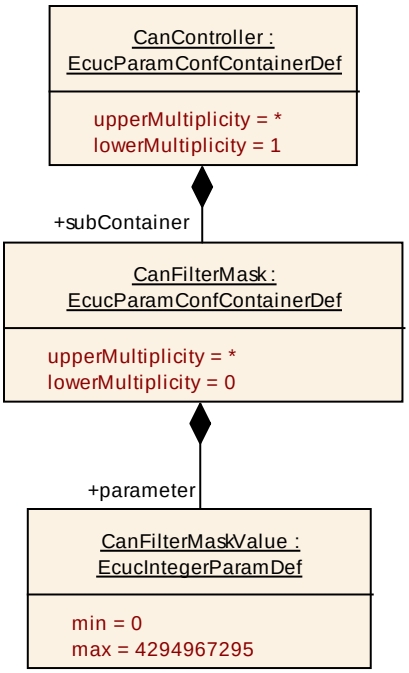


Figure 10-6: Can Filter Mask Configuration Layout

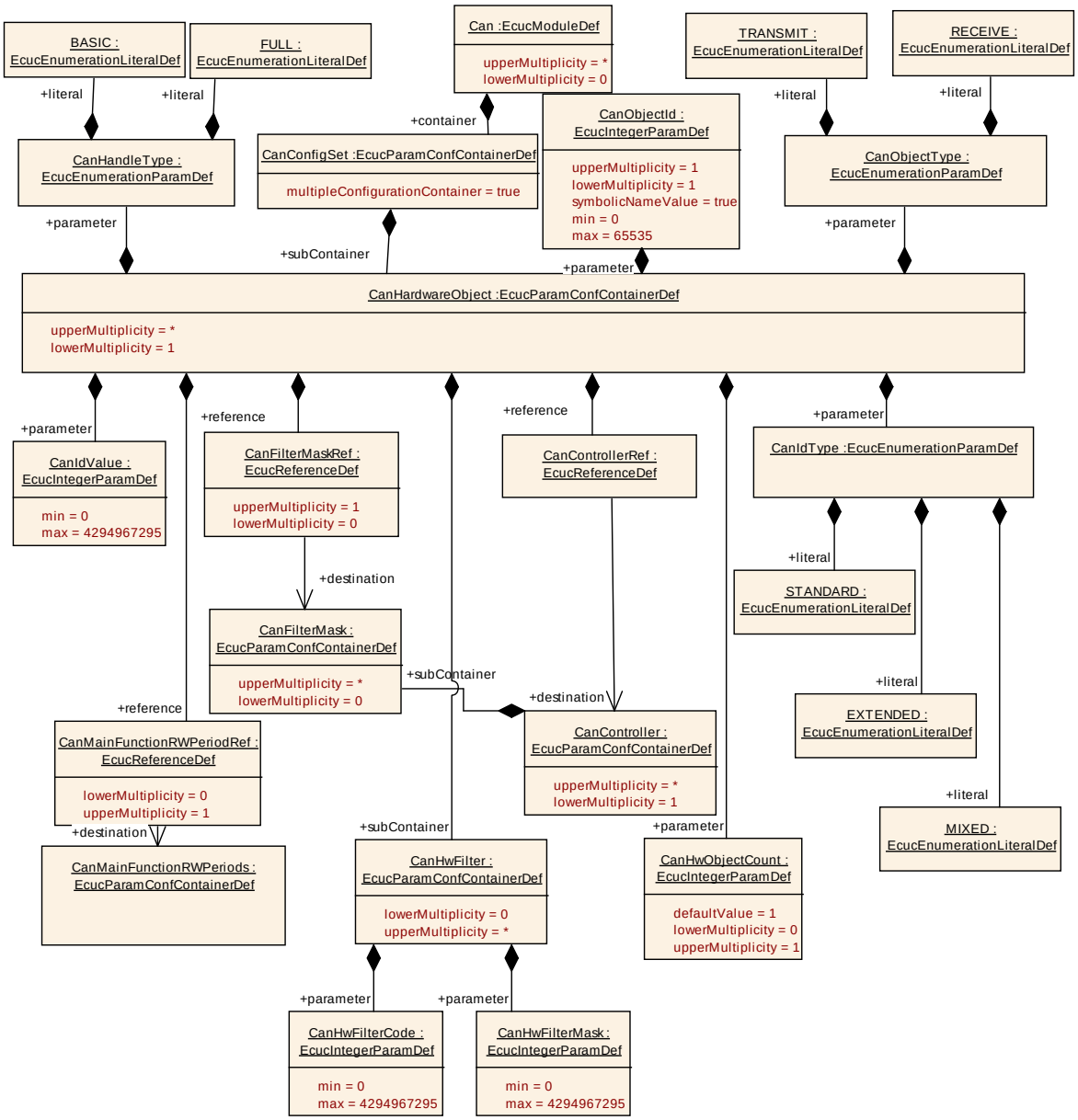


Figure 10-7: Can Hardware Object Configuration Layout

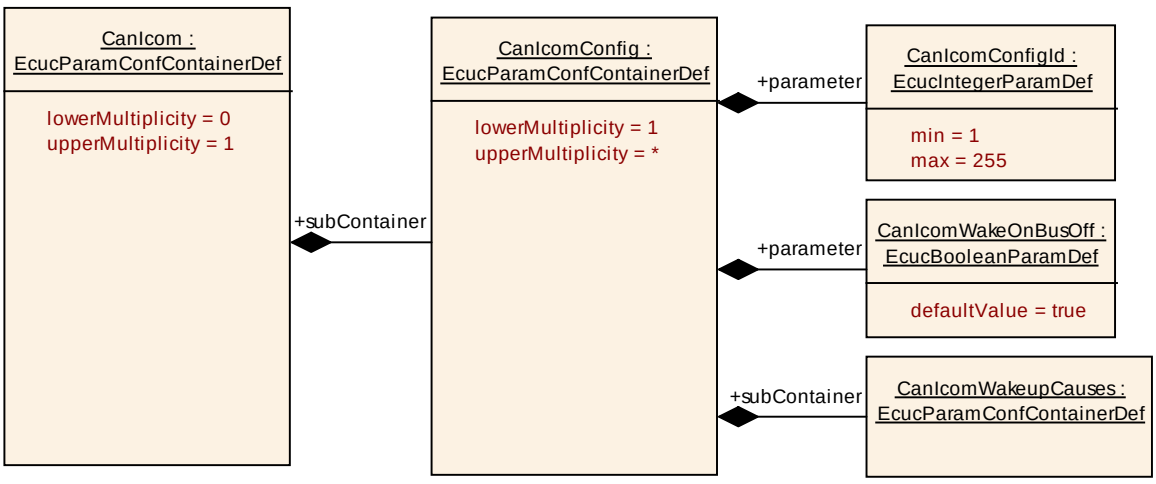


Figure 10-8: CanICOM Layout

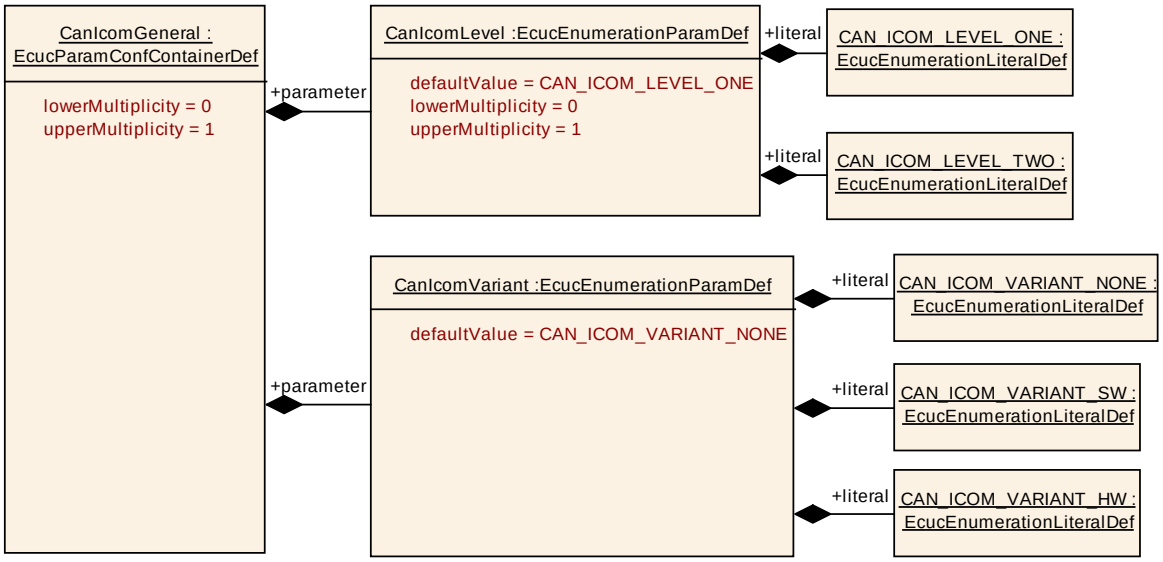


Figure 10-9: CanICOM General Configuration Layout

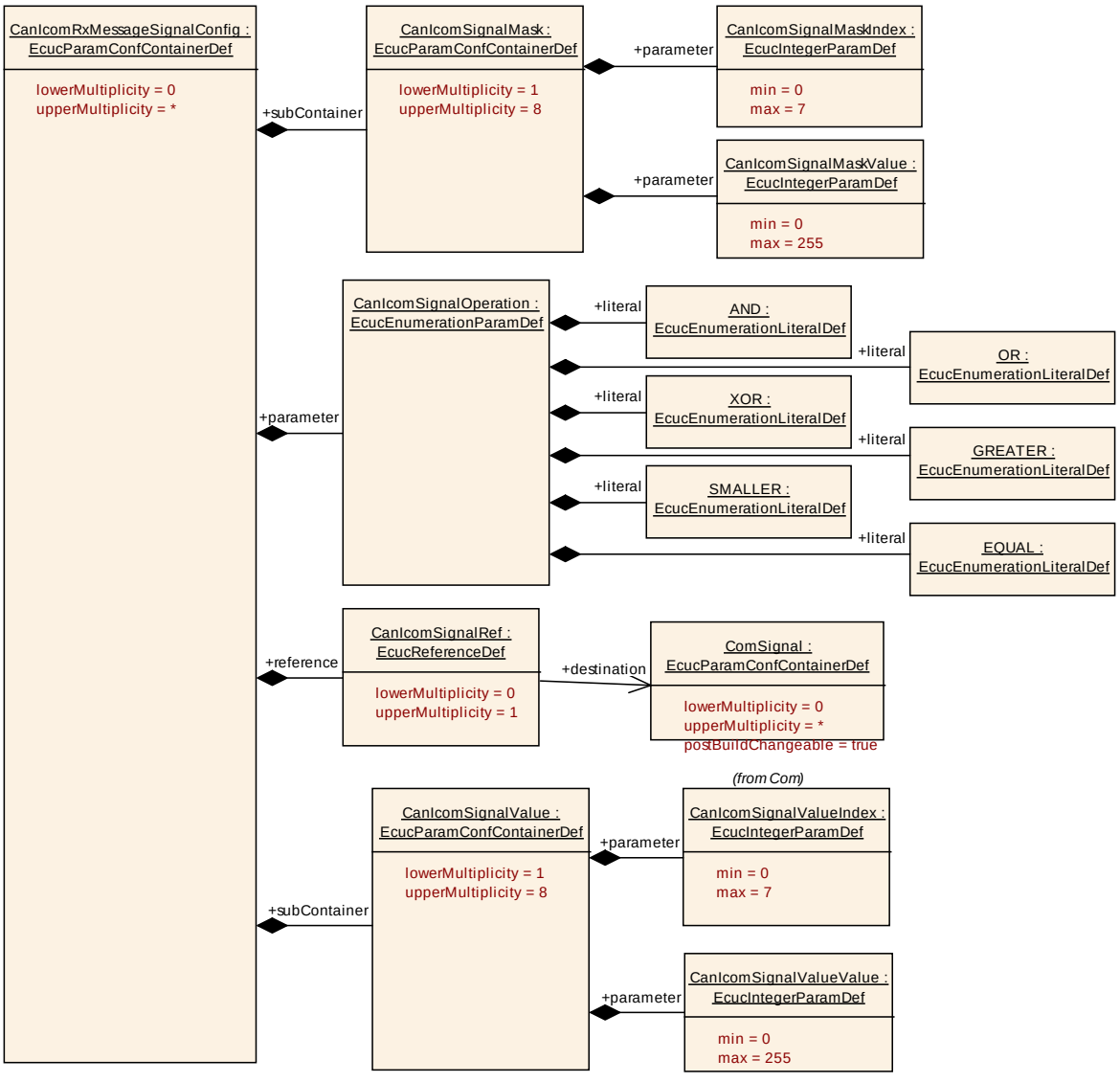


Figure 10-10: CanIcomRxMessageSignal Configuration Layout

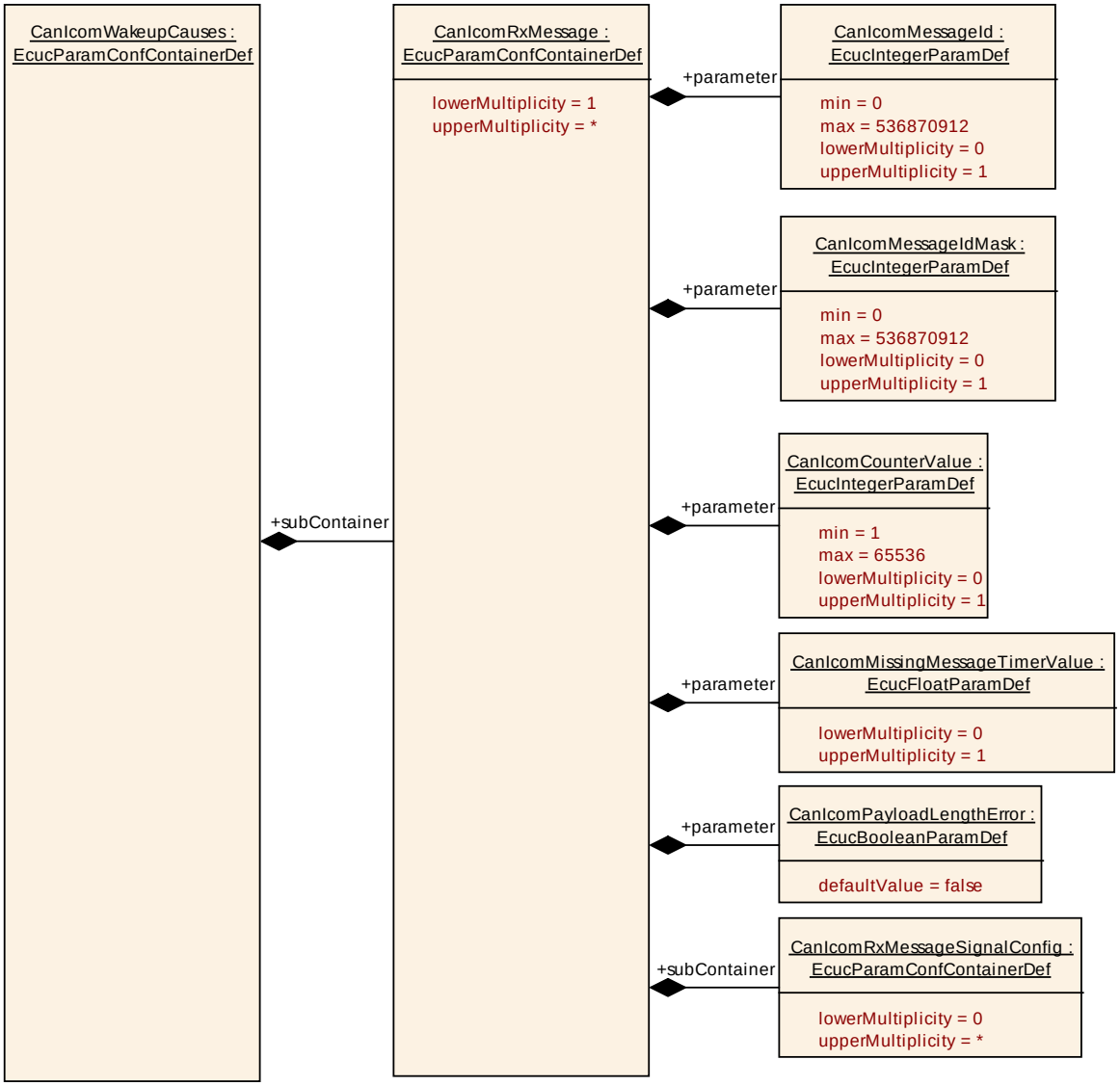


Figure 10-11: CanIcomWakeupCauses Configuration Layout

10.2.2 Can

Module Name	*Can*
Module Description	This container holds the configuration of a single CAN Driver.

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanConfigSet	1	This is the multiple configuration set container for CAN Driver
CanGeneral	1	This container contains the parameters related each CAN Driver Unit.

10.2.3 CanGeneral

SWS Item	**ECUC_Can_00328 :**
Container Name	CanGeneral{CanDriverGeneralConfiguration}
Description	This container contains the parameters related each CAN Driver Unit.
Configuration Parameters	

SWS Item	**ECUC_Can_00436 : (Obsolete)**		
Name	CanChangeBaudrateApi {CAN_CHANGE_BAUDRATE_API}		
Description	The support of the Can_ChangeBaudrate API is optional. If this parameter is set to true the Can_ChangeBaudrate API shall be supported. Otherwise the API is not supported. Please note that the Can_ChangeBaudrate API and this parameter are deprecated and will be removed in future. **Tags:** atp.Status=obsolete atp.StatusRevisionBegin=4.1.1		
Multiplicity	0..1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Can_00064 :**		
Name	CanDevErrorDetection {CAN_DEV_ERROR_DETECT}		
Description	Switches the Development Error Detection and Notification ON or OFF.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00069 :**
Name	CanHardwareCancellation {CAN_HW_TRANSMIT_CANCELLATION}
Description	Specifies if hardware cancellation shall be supported.ON or OFF
Multiplicity	1

Type	EcucBooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: CanIf module is configured to support hardware cancellation		

SWS Item	**ECUC_Can_00378 :**		
Name	CanIdenticalIdCancellation {CAN_IDENTICAL_ID_CANCELLATION}		
Description	Enables/disables cancellation of pending PDUs with identical ID.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00320 :**		
Name	CanIndex		
Description	Specifies the InstanceId of this module instance. If only one instance is present it shall have the Id 0.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 255		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00434 :**		
Name	CanLPduReceiveCalloutFunction		
Description	This parameter defines the existence and the name of a callout function that is called after a successful reception of a received CAN Rx L-PDU. If this parameter is omitted no callout shall take place.		
Multiplicity	0..1		
Type	EcucFunctionNameDef		
Default value	--		
maxLength	--		
minLength	--		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00355 :**		
Name	CanMainFunctionBusoffPeriod		
Description	This parameter describes the period for cyclic call to Can_MainFunction_Busoff. Unit is seconds.		
Multiplicity	0..1		
Type	EcucFloatParamDef		
Range	0.001 .. 65.535		

Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**ECUC_Can_00376 :**		
Name	CanMainFunctionModePeriod		
Description	This parameter describes the period for cyclic call to Can_MainFunction_Mode. Unit is seconds.		
Multiplicity	1		
Type	EcucFloatParamDef		
Range	0.001 .. 65.535		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**ECUC_Can_00357 :**		
Name	CanMainFunctionWakeupPeriod		
Description	This parameter describes the period for cyclic call to Can_MainFunction_Wakeup. Unit is seconds.		
Multiplicity	0..1		
Type	EcucFloatParamDef		
Range	0.001 .. 65.535		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**ECUC_Can_00095 :**		
Name	CanMultiplexedTransmission {CAN_MULTIPLEXED_TRANSMISSION}		
Description	Specifies if multiplexed transmission shall be supported.ON or OFF		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: CAN Hardware Unit supports multiplexed transmission		

SWS Item	**ECUC_Can_00483 :**		
Name	CanPublicIcomSupport {CAN_PUBLIC_ICOM_SUPPORT}		
Description	Selects support of Pretended Network features in Can driver. True: Enabled False: Disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Can_00482 :**		
Name	CanSetBaudrateApi {CAN_SET_BAUDRATE_API}		
Description	The support of the Can_SetBaudrate API is optional. If this parameter is set to true the Can_SetBaudrate API shall be supported. Otherwise the API is not supported.		
Multiplicity	0..1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Can_00113 :**		
Name	CanTimeoutDuration {CAN_TIMEOUT_DURATION}		
Description	Specifies the maximum time for blocking function until a timeout is detected. Unit is seconds.		
Multiplicity	1		
Type	EcucFloatParamDef		
Range	1E-6 .. 65.535		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00106 :**		
Name	CanVersionInfoApi {CAN_VERSION_INFO_API}		
Description	Switches the Can_GetVersionInfo() API ON or OFF.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00431 :**		
Name	CanCounterRef		
Description	This parameter contains a reference to the counter, which is used by the CAN driver.		
Multiplicity	1		
Type	Reference to [OsCounter]		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00430 :**		
Name	CanSupportTTCANRef		
Description	The parameter refers to CanIfSupportTTCAN parameter in the CAN Interface Module configuration. The CanIfSupportTTCAN parameter defines whether TTCAN is supported.		
Multiplicity	1		
Type	Reference to [CanIfPrivateCfg]		
ConfigurationClass	***Pre-compile time***	X	All Variants

	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanIcomGeneral	0..1	This container contains the general configuration parameters of the ICOM Configuration.
CanMainFunctionRWPeriods	1	--

10.2.4 CanController

SWS Item	**ECUC_Can_00354 :**
Container Name	CanController{CanController}
Description	This container contains the configuration parameters of the CAN controller(s).
Configuration Parameters	

SWS Item	**ECUC_Can_00314 :**		
Name	CanBusoffProcessing {CAN_BUSOFF_PROCESSING}		
Description	Enables / disables API Can_MainFunction_BusOff() for handling busoff events in polling mode.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	INTERRUPT	Interrupt Mode of operation.	
	POLLING	Polling Mode of operation.	
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00315 :**		
Name	CanControllerActivation {CAN_CONTROLLER_ACTIVATION}		
Description	Defines if a CAN controller is used in the configuration.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00382 :**		
Name	CanControllerBaseAddress {CAN_CONTROLLER_BASE_ADDRESS}		
Description	Specifies the CAN controller base address.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 4294967295		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00316 :**		
Name	CanControllerId {CAN_DRIVER_CONTROLLER_ID}		
Description	This parameter provides the controller ID which is unique in a given CAN Driver. The value for this parameter starts with 0 and continue without any gaps.		
Multiplicity	1		
Type	EcucIntegerParamDef (Symbolic Name generated for this parameter)		
Range	0 .. 255		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Can_00317 :**		
Name	CanRxProcessing {CAN_RX_PROCESSING}		
Description	Enables / disables API Can_MainFunction_Read() for handling PDU reception events in polling mode.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	INTERRUPT	Interrupt Mode of operation.	
	POLLING	Polling Mode of operation.	
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00318 :**		
Name	CanTxProcessing {CAN_TX_PROCESSING}		
Description	Enables / disables API Can_MainFunction_Write() for handling PDU transmission events in polling mode.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	INTERRUPT	Interrupt Mode of operation.	
	POLLING	Polling Mode of operation.	
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00466 :**		
Name	CanWakeupFunctionalityAPI		
Description	Adds / removes the service Can_CheckWakeup() from the code. True: Can_CheckWakeup can be used. False: Can_CheckWakeup cannot be used.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local dependency: H/W should support the wakeup functionality to enable this parameter.		

SWS Item	**ECUC_Can_00319 :**

Name	CanWakeupProcessing {CAN_WAKEUP_PROCESSING}		
Description	Enables / disables API Can_MainFunction_Wakeup() for handling wakeup events in polling mode.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	INTERRUPT	Interrupt Mode of operation.	
	POLLING	Polling Mode of operation.	
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00330 :**		
Name	CanWakeupSupport {CAN_WAKEUP_SUPPORT}		
Description	CAN driver support for wakeup over CAN Bus.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**ECUC_Can_00435 :**		
Name	CanControllerDefaultBaudrate {CAN_CONTROLLER_DEFAULT_BAUDRATE}		
Description	Reference to baudrate configuration container configured for the Can Controller.		
Multiplicity	1		
Type	Reference to [CanControllerBaudrateConfig]		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00313 :**		
Name	CanCpuClockRef {CAN_CPU_CLOCK_REFERENCE}		
Description	Reference to the CPU clock configuration, which is set in the MCU driver configuration		
Multiplicity	1		
Type	Reference to [McuClockReferencePoint]		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00359 :**		
Name	CanWakeupSourceRef		
Description	This parameter contains a reference to the Wakeup Source for this controller as defined in the ECU State Manager. Implementation Type: reference to EcuM_WakeupSourceType		
Multiplicity	0..1		
Type	Symbolic name reference to [EcuMWakeupSource]		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	

Scope / Dependency	scope: local

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanControllerBaudrateConfig	1..*	This container contains bit timing related configuration parameters of the CAN controller(s).
CanFilterMask	0..*	This container contains the configuration (parameters) of the CAN Filter Mask(s). This container is set to obsolete. It is replaced by CanHwFilterMask
CanTTController	0..1	CanTTController is specified in the SWS TTCAN and contains the configuration parameters of the TTCAN controller(s) (which are needed in addition to the configuration parameters of the CAN controller(s)). This container is only included and valid if TTCAN is supported by the controller, enabled (see CanSupportTTCANRef, ECUC_Can_00430), and used.

10.2.5 CanControllerBaudrateConfig

SWS Item	**ECUC_Can_00387 :**
Container Name	CanControllerBaudrateConfig{CanControllerBaudrateConfig}
Description	This container contains bit timing related configuration parameters of the CAN controller(s).
Configuration Parameters	

SWS Item	**ECUC_Can_00005 :**		
Name	CanControllerBaudRate {CAN_CONTROLLER_BAUD_RATE}		
Description	Specifies the baudrate of the controller in kbps.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 2000		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00471 :**		
Name	CanControllerBaudRateConfigID {CAN_CONTROLLER_BAUD_RATE_CONFIG_ID}		
Description	Uniquely identifies a specific baud rate configuration. This ID is used by SetBaudrate API.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 65535		
Default value	0		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU dependency: CanSetBaudrateApi		

SWS Item	**ECUC_Can_00073 :**

Name	CanControllerPropSeg {CAN_CONTROLLER_PROP_SEG}		
Description	Specifies propagation delay in time quantas.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 255		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00074 :**		
Name	CanControllerSeg1 {CAN_CONTROLLER_PHASE_SEG1}		
Description	Specifies phase segment 1 in time quantas.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 255		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00075 :**		
Name	CanControllerSeg2 {CAN_CONTROLLER_PHASE_SEG2}		
Description	Specifies phase segment 2 in time quantas.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 255		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00383 :**		
Name	CanControllerSyncJumpWidth {CAN_CONTROLLER_SJW}		
Description	Specifies the synchronization jump width for the controller in time quantas.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 255		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanControllerFdBaudrateConfig	0..1	This optional container contains bit timing related configuration parameters of the CAN controller(s) for payload and CRC of a CAN FD frame. If this container exists the controller supports CAN FD frames.

10.2.6 CanControllerFdBaudrateConfig

SWS Item	**ECUC_Can_00473 :**		
Container Name	CanControllerFdBaudrateConfig		
Description	This optional container contains bit timing related configuration parameters of the CAN controller(s) for payload and CRC of a CAN FD frame. If this container exists the controller supports CAN FD frames.		
Configuration Parameters			

SWS Item	**ECUC_Can_00481 :**		
Name	CanControllerFdBaudRate {CAN_CONTROLLER_BAUD_RATE}		
Description	Specifies the data segment baud rate of the controller in kbps.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 16000		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00476 :**		
Name	CanControllerPropSeg {CAN_CONTROLLER_PROP_SEG}		
Description	Specifies propagation delay in time quantas.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 255		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00477 :**		
Name	CanControllerSeg1 {CAN_CONTROLLER_PHASE_SEG1}		
Description	Specifies phase segment 1 in time quantas.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 255		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00478 :**		
Name	CanControllerSeg2 {CAN_CONTROLLER_PHASE_SEG2}		
Description	Specifies phase segment 2 in time quantas.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 255		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

SWS Item	ECUC_Can_00479 :		
Name	CanControllerSyncJumpWidth {CAN_CONTROLLER_SJW}		
Description	Specifies the synchronization jump width for the controller in time quantas.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 255		
Default value	--		
ConfigurationClass	Pre-compile time	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

SWS Item	ECUC_Can_00480 :		
Name	CanControllerTrcvDelayCompensationOffset {CAN_CONTROLLER_TDC_OFFSET}		
Description	Specifies the Transceiver Delay Compensation Offset in ns. If not specified Transceiver Delay Compensation is disabled.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 400		
Default value	--		
ConfigurationClass	Pre-compile time	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

SWS Item	ECUC_Can_00475 :		
Name	CanControllerTxBitRateSwitch {CAN_CONTROLLER_TX_BRS}		
Description	Specifies if the bit rate switching shall be used for transmissions. If FALSE: CAN FD frames shall be sent without bit rate switching.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	true		
ConfigurationClass	Pre-compile time	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

No Included Containers

10.2.7 CanHardwareObject

SWS Item	ECUC_Can_00324 :
Container Name	CanHardwareObject{CanHardwareObject}
Description	This container contains the configuration (parameters) of CAN Hardware Objects.
Configuration Parameters	

SWS Item	ECUC_Can_00323 :
Name	CanHandleType {CAN_HANDLE_TYPE}
Description	Specifies the type (Full-CAN or Basic-CAN) of a hardware object.
Multiplicity	1

Type	EcucEnumerationParamDef		
Range	BASIC	For several L-PDUs are hadled by the hardware object	
	FULL	For only one L-PDU (identifier) is handled by the hardware object	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU dependency: This configuration element is used as information for the CAN Interface only. The relevant CAN driver configuration is done with the filter mask and identifier.		

SWS Item	**ECUC_Can_00467 :**		
Name	CanHwObjectCount		
Description	Number of hardware objects used to implement one HOH. In case of a HRH this parameter defines the number of elements in the hardware FIFO or the number of shadow buffers, in case of a HTH it defines the number of hardware objects used for multiplexed transmission or for a hardware FIFO used by a FullCAN HTH.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 65535		
Default value	1		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Can_00065 :**		
Name	CanIdType {CAN_ID_TYPE}		
Description	Specifies whether the IdValue is of type - standard identifier - extended identifier - mixed mode ImplementationType: Can_IdType		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	EXTENDED	All the CANIDs are of type extended only (29 bit).	
	MIXED	The type of CANIDs can be both Standard or Extended.	
	STANDARD	All the CANIDs are of type standard only (11bit).	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Can_00325 : (Obsolete. Use CanHwFilterCode instead.)**
Name	CanIdValue {CAN_ID_VALUE}
Description	Specifies (together with the filter mask) the identifiers range that passes the hardware filter. This parameter is set to obsolete and will be replaced by CanHwFilterCode

	Tags: atp.Status=obsolete atp.StatusComment=This parameter is replaced by CanHwFilterCode atp.StatusRevisionBegin=4.1.1		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 4294967295		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Can_00326 :**		
Name	CanObjectId {CAN_OBJECT_HANDLE_ID}		
Description	Holds the handle ID of HRH or HTH. The value of this parameter is unique in a given CAN Driver, and it should start with 0 and continue without any gaps. The HRH and HTH Ids share a common ID range. Example: HRH0-0, HRH1-1, HTH0-2, HTH1-3		
Multiplicity	1		
Type	EcucIntegerParamDef (Symbolic Name generated for this parameter)		
Range	0 .. 65535		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Can_00327 :**		
Name	CanObjectType {CAN_OBJECT_TYPE}		
Description	Specifies if the HardwareObject is used as Transmit or as Receive object		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	RECEIVE		Receive HOH
	TRANSMIT		Transmit HOH
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Can_00322 :**		
Name	CanControllerRef {CAN_CONTROLLER_REFERENCE}		
Description	Reference to CAN Controller to which the HOH is associated to.		
Multiplicity	1		
Type	Reference to [CanController]		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00321 : (Obsolete. Use CanHwFilterMask instead.)**
Name	CanFilterMaskRef {CAN_MASK_REFERENCE}
Description	Reference to the filter mask that is used for hardware filtering together with the CAN_ID_VALUE. Different CanHardwareObjects with different CanIdTypes (STANDARD,

	MIXED, EXTENDED) can share the same CanFilterMask (i.e., the CanFilterMaskRef parameters of these CanHardwareObjects reference the very same CanFilterMask container). This shall be allowed and must be supported by the configuration generators. The CanFilterMaskRef is omitted for 1) CanHardwareObjects with CanObjectType set to TRANSMIT 2) CanHardwareObjects with CanObjectType set to RECEIVE if only a single Can ID shall be received via this CanHardwareObjects (i.e., exact match with CanIdValue) This parameter is set to obsolete and is replaced by CanHwFilterMask **Tags:** atp.Status=obsolete atp.StatusComment=This reference is replaced by CanHwFilterMask atp.StatusRevisionBegin=4.1.1		
Multiplicity	0..1		
Type	Reference to [CanFilterMask]		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00438 :**		
Name	CanMainFunctionRWPeriodRef {CAN_CONTROLLER_REFERENCE}		
Description	Reference to CanMainFunctionReadPeriod and CanMainFunctionWritePeriod		
Multiplicity	0..1		
Type	Reference to [CanMainFunctionRWPeriods]		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: local		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanHwFilter	0..*	This container is only valid for HRHs and contains the configuration (parameters) of one hardware filter.
CanTTHardwareObjectTrigger	0..*	CanTTHardwareObjectTrigger is specified in the SWS TTCAN and contains the configuration (parameters) of TTCAN triggers for Hardware Objects, which are additional to the configuration (parameters) of CAN Hardware Objects. This container is only included and valid if TTCAN is supported by the controller and, enabled (see CanSupportTTCANRef, ECUC_Can_00430), and used.

10.2.8 CanHwFilter

SWS Item	**ECUC_Can_00468 :**
Container Name	CanHwFilter
Description	This container is only valid for HRHs and contains the configuration (parameters) of one hardware filter.
Configuration Parameters	

SWS Item	**ECUC_Can_00469 :**		
Name	CanHwFilterCode		
Description	Specifies (together with the filter mask) the identifiers range that passes the hardware filter.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 4294967295		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency			

SWS Item	**ECUC_Can_00470 :**		
Name	CanHwFilterMask		
Description	Describes a mask for hardware-based filtering of CAN identifiers. The CAN identifiers of incoming messages are masked with the appropriate CanFilterMaskValue. Bits holding a 0 mean don't care, i.e. do not compare the message's identifier in the respective bit position. The mask shall be build by filling with leading 0. In case of CanIdType EXTENDED or MIXED a 29 bit mask shall be build. In case of CanIdType STANDARD a 11 bit mask shall be build		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 4294967295		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	dependency: The filter mask settings must be known by the CanIf configuration for optimization of the SW filters.		

No Included Containers

10.2.9 CanConfigSet

SWS Item	**ECUC_Can_00343 :**
Container Name	CanConfigSet [Multi Config Container]
Description	This is the multiple configuration set container for CAN Driver
Configuration Parameters	

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanController	1..*	This container contains the configuration parameters of the CAN controller(s).
CanHardwareObject	1..*	This container contains the configuration (parameters) of CAN Hardware Objects.
CanIcom	0..1	This container contains the parameters for configuring pretended networking

10.2.10 CanMainFunctionRWPeriods

SWS Item	**ECUC_Can_00437 :**		
Container Name	CanMainFunctionRWPeriods{CAN_MAIN_FUNCTION_RWPERIODS}		
Description	--		
Configuration Parameters			

SWS Item	**ECUC_Can_00356 :**		
Name	CanMainFunctionReadPeriod		
Description	This parameter describes the period for cyclic call to Can_MainFunction_Read. Unit is seconds. Different poll-cycles will be configurable if more than one CanMainFunctionReadPeriod is configured. In this case multiple Can_MainFunction_Read() will be provided by the CAN Driver module.		
Multiplicity	0..*		
Type	EcucFloatParamDef		
Range	0.001 .. 65.535		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Can_00358 :**		
Name	CanMainFunctionWritePeriod		
Description	This parameter describes the period for cyclic call to Can_MainFunction_Write. Unit is seconds. Different poll-cycles will be configurable if more than one CanMainFunctionWritePeriod is configured. In this case multiple Can_MainFunction_Write() will be provided by the CAN Driver module.		
Multiplicity	0..*		
Type	EcucFloatParamDef		
Range	0.001 .. 65.535		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

No Included Containers

10.2.11 CanIcom

SWS Item	**ECUC_Can_00440 :**
Container Name	CanIcom
Description	This container contains the parameters for configuring pretended networking
Configuration Parameters	

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanIcomConfig	1..*	This container contains the configuration parameters of the ICOM Configuration.

10.2.12 CanIcomConfig

SWS Item	**ECUC_Can_00459 :**		
Container Name	CanIcomConfig{CAN_ICOM_CONFIG}		
Description	This container contains the configuration parameters of the ICOM Configuration.		
Configuration Parameters			

SWS Item	**ECUC_Can_00441 :**		
Name	CanIcomConfigId {CAN_ICOM_CONFIG_ID}		
Description	This parameter identifies the ID of the ICOM configuration.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	1 .. 255		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Can_00442 :**		
Name	CanIcomWakeOnBusOff {CAN_ICOM_WAKE_ON_BUS_OFF}		
Description	This parameter defines that the MCU shall wake if the bus off is detected or not.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	true		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanIcomWakeupCauses	1	This container contains the configuration parameters of the wakeup causes to leave the power saving mode.

10.2.13 CanIcomGeneral

SWS Item	**ECUC_Can_00444 :**		
Container Name	CanIcomGeneral{CAN_ICOM_GENERAL}		
Description	This container contains the general configuration parameters of the ICOM Configuration.		
Configuration Parameters			

SWS Item	**ECUC_Can_00445 :**	
Name	CanIcomLevel {CAN_ICOM_LEVEL}	
Description	Defines the level of Pretended Networking. This parameter is reserved for future implementations (Pretended Networking level 2).	
Multiplicity	0..1	
Type	EcucEnumerationParamDef	
Range	CAN_ICOM_LEVEL_ONE	-- (default)

	CAN_ICOM_LEVEL_TWO	--	
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Can_00446 :**		
Name	CanIcomVariant {CAN_ICOM_VARIANT}		
Description	Defines the variant, which is supported by this CanController		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	CAN_ICOM_VARIANT_HW	--	
	CAN_ICOM_VARIANT_NONE	-- (default)	
	CAN_ICOM_VARIANT_SW	--	
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

No Included Containers

10.2.14 CanIcomRxMessage

SWS Item	**ECUC_Can_00447 :**
Container Name	CanIcomRxMessage{CAN_ICOM_RX_MESSAGE}
Description	This container contains the configuration parameters for the wakeup causes for matching received messages. It has to be configured as often as received messages are defined as wakeup cause. constraint: For all CanIcomRxMessage instances the Message IDs which are defined in CanIcomMessageId and in CanIcomRxMessageRange shall never overlap. Also the ranges of CanIcomRxMessageRange shall not overlap each other.
Configuration Parameters	

SWS Item	**ECUC_Can_00448 :**		
Name	CanIcomCounterValue {CAN_ICOM_COUNTER_VALUE}		
Description	This parameter defines that the MCU shall wake if the message with the ID is received n times on the communication channel.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	1 .. 65536		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Can_00449 :**
Name	CanIcomMessageId {CAN_ICOM_MESSAGE_ID}
Description	This parameter defines the message ID the wakeup causes of this CanIcomRxMessage are configured for. In addition a mask can be defined,CanIcomMessageIdMask) in that case it is possible to define a

	range of rx messages,which can create and wakeup.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 536870912		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: CanIcomMessageId and CanIcomRxMessageRange shall not exist both within one CanIcomRxMessage instance.		

SWS Item	**ECUC_Can_00465 :**		
Name	CanIcomMessageIdMask {CAN_ICOM_MESSAGE_ID}		
Description	Describes a mask for filtering of CAN identifiers. The CAN identifiers of incoming messages are masked with the appropriate CanIcomMessagIdMask. Bits holding a 0 mean don't care, i.e. do not compare the message's identifier in the respective bit position. The mask shall be build by filling with leading 0.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 536870912		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: CanIcomMessageId and CanIcomRxMessageRange shall not exist both within one CanIcomRxMessage instance.		

SWS Item	**ECUC_Can_00450 :**		
Name	CanIcomMissingMessageTimerValue {CAN_ICOM_MISSING_MESSAGE_TIMER_VALUE}		
Description	This parameter defines that the MCU shall wake if the message with the ID is not received for a specific time in ms on the communication channel.		
Multiplicity	0..1		
Type	EcucFloatParamDef		
Range	-INF .. INF		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Can_00451 :**		
Name	CanIcomPayloadLengthError {CAN_ICOM_PAYLOAD_LENGTH_ERROR}		
Description	This parameter defines that the MCU shall wake if a payload error occurs		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

Included Containers

Container Name	Multiplicity	Scope / Dependency
CanIcomRxMessageSignalConfig	0..*	This container contains the configuration parameters for the wakeup causes for matching signals. It has to be configured as often as a signal is defined as wakeup cause. If at least one the Signal conditions evaluate to true, the whole wakeup condition is considered to be true. All instances of this container refer to the same frame/pdu

10.2.15 CanIcomRxMessageSignalConfig

SWS Item	ECUC_Can_00452 :
Container Name	CanIcomRxMessageSignalConfig{CAN_ICOM_RX_MESSAGE_SIGNAL_CONFIG}
Description	This container contains the configuration parameters for the wakeup causes for matching signals. It has to be configured as often as a signal is defined as wakeup cause. If at least one the Signal conditions evaluate to true, the whole wakeup condition is considered to be true. All instances of this container refer to the same frame/pdu
Configuration Parameters	

SWS Item	ECUC_Can_00462 :		
Name	CanIcomSignalOperation {CAN_ICOM_SIGNAL_OPERATION}		
Description	This parameter defines the operation, which shall be used to verify the payload data. AND: The masked payload via CanIcomSignalMask AND CanIcomSignalValue must be TRUE (logical AND) OR: The masked payload via CanIcomSignalMask OR CanIcomSignalValue must be TRUE (logical OR) XOR: The masked payload via CanIcomSignalMask XOR CanIcomSignalValue must be TRUE (logical XOR) SMALLER: The masked payload via CanIcomSignalMask must be strictly smaller than CanIcomSignalValue EQUAL: The masked payload via CanIcomSignalMask must beequal to CanIcomSignalValue GREATER: The masked payload via CanIcomSignalMask must be strictly greater than CanIcomSignalValue GREATER and SMALLER can only be applied to unsigned integer values. It is known that AND and EQUAL can be considered the same, but for clarification reason both are kept to show that AND is taken for the arithmetical and EQUAL is taken for the logical operation.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	AND	--	
	EQUAL	--	
	GREATER	--	
	OR	--	
	SMALLER	--	
	XOR	--	
ConfigurationClass	Pre-compile time	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	ECUC_Can_00456 :
Name	CanIcomSignalRef {CAN_ICOM_SIGNAL_REF}
Description	This parameter defines a reference to the signal which shall be checked additional to the message id. This reference is used for documentation to define which ComSignal originates this filter setting. All signals being refered by this reference shall

	point to the same PDU.		
Multiplicity	0..1		
Type	Reference to [ComSignal]		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanIcomSignalMask	1..8	This container defines the configuration for the signal mask to extract the signal out of the payload of the received message. The mask shall be logical ANDed with the received payload. The result shall be used with the CanIcomSignalOperation with the specified parameters in CanIcomSignalValue
CanIcomSignalValue	1..8	This container defines the configuration for the signal value which shall be used to compare with the signal in the received message.

10.2.16 CanIcomSignalMask

SWS Item	**ECUC_Can_00460 :**
Container Name	CanIcomSignalMask{CAN_ICOM_SIGNAL_MASK}
Description	This container defines the configuration for the signal mask to extract the signal out of the payload of the received message. The mask shall be logical ANDed with the received payload. The result shall be used with the CanIcomSignalOperation with the specified parameters in CanIcomSignalValue
Configuration Parameters	

SWS Item	**ECUC_Can_00461 :**		
Name	CanIcomSignalMaskIndex {CAN_ICOM_SIGNAL_MASK_INDEX}		
Description	This parameter defines the byte number of the payload data to identifiy the signal. The signal can be at least one bit and in maximum 8 bits. If the signal is greater than 8 bits more than one CanIcomSignalMaskIndex must be defined. The CanIcomSignalMaskIndex works in combination with the CanIcomSignalMaskValue.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 7		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Can_00463 :**
Name	CanIcomSignalMaskValue {CAN_ICOM_SIGNAL_MASK_VALUE}
Description	This parameter shall be used to mask a signal in the payload of a CAN message. The mask is logical AND with the payload. The result will be used in combination of the operations defined in CanIcomSignalOperation with the CanIcomSignalValue.
Multiplicity	1

Type	EcucIntegerParamDef		
Range	0 .. 255		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

No Included Containers

10.2.17 CanIcomSignalValue

SWS Item	**ECUC_Can_00457 :**
Container Name	CanIcomSignalValue{CAN_ICOM_SIGNAL_VALUE}
Description	This container defines the configuration for the signal value which shall be used to compare with the signal in the received message.
Configuration Parameters	

SWS Item	**ECUC_Can_00458 :**		
Name	CanIcomSignalValueIndex {CAN_ICOM_SIGNAL_VALUE_INDEX}		
Description	This parameter defines the byte number of the value, which shall be used to compare with the received signal in combination with the CanIcomSignalOperation.The signal can be at least one bit and in maximum 8 bits. If the signal is greater than 8 bits more than one CanIcomSignalValueIndex must be defined. The CanIcomSignalValueIndex works in combination with the CanIcomSignalValueValue.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 7		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Can_00464 :**		
Name	CanIcomSignalValueValue {CAN_ICOM_SIGNAL_VALUE_VALUE}		
Description	This parameter shall be used to define a signal which shall be used in combination with the received signal in the payload of a CAN message. The parameter will be used in combination of the operations defined in CanIcomSignalOperation with the result of the payload and the CanIcomSignalMask.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 255		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

No Included Containers

11 Not applicable requirements

[SWS_Can_00999]「 These requirements are not applicable to this specification. 」(SRS_BSW_00170, SRS_BSW_00383, SRS_BSW_00395, SRS_BSW_00397, SRS_BSW_00398, SRS_BSW_00399, SRS_BSW_00400, SRS_BSW_00168, SRS_BSW_00423, SRS_BSW_00424, SRS_BSW_00425, SRS_BSW_00426, SRS_BSW_00427, SRS_BSW_00429, SRS_BSW_00433, SRS_BSW_00336, SRS_BSW_00422, SRS_BSW_00417, SRS_BSW_00409, SRS_BSW_00455, SRS_BSW_00162, SRS_BSW_00415, SRS_BSW_00325, SRS_BSW_00326, SRS_BSW_00342, SRS_BSW_00453, SRS_BSW_00413, SRS_BSW_00307, SRS_BSW_00447, SRS_BSW_00353, SRS_BSW_00361, SRS_BSW_00439, SRS_BSW_00449, SRS_BSW_00378, SRS_BSW_00359, SRS_BSW_00440, BSW00443, BSW00444, BSW00445, BSW00446, SRS_SPAL_12163, SRS_SPAL_12462, SRS_SPAL_12068, SRS_SPAL_12064, SRS_Can_01125, SRS_Can_01126)