Document Title	Specification of CAN Driver
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	011
Document Classification	Standard

Document Version	4.0.0
Document Status	Final
Part of Release	4.0
Revision	3

Document Change History

Date	Version	Changed by	Change Description
02.11.2011	4.0.0	AUTOSAR Administration	• Added CAN461 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 CAN321_Conf • Changed Can_SetControllerMode in CAN370 to Can_Mainfunction_Mode • Added CanControllerDefaultBaudrate parameter • Updated description of CAN279 • Updated description of CAN321 • Added CAN445, CAN446 and CAN447 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 CAN111 to correct the "Version Checking" information • Added new requirements CAN435 to CAN440 to introduce Can_GeneralTypes.h. • Added new requirements CAN441 and CAN442 to introduce multiple poll cycles • Added new requirements CAN443 and CAN444 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 5.2) • Removed return value CAN_WAKEUP in function Can_SetControllerMode • Replaced by CAN_NOT_OK • Renamed CanIf_ControllerWakeup to CanIf_SetWakeupEvent • Reworked development errors (chapter 7.10) • 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 BSW12092).

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.

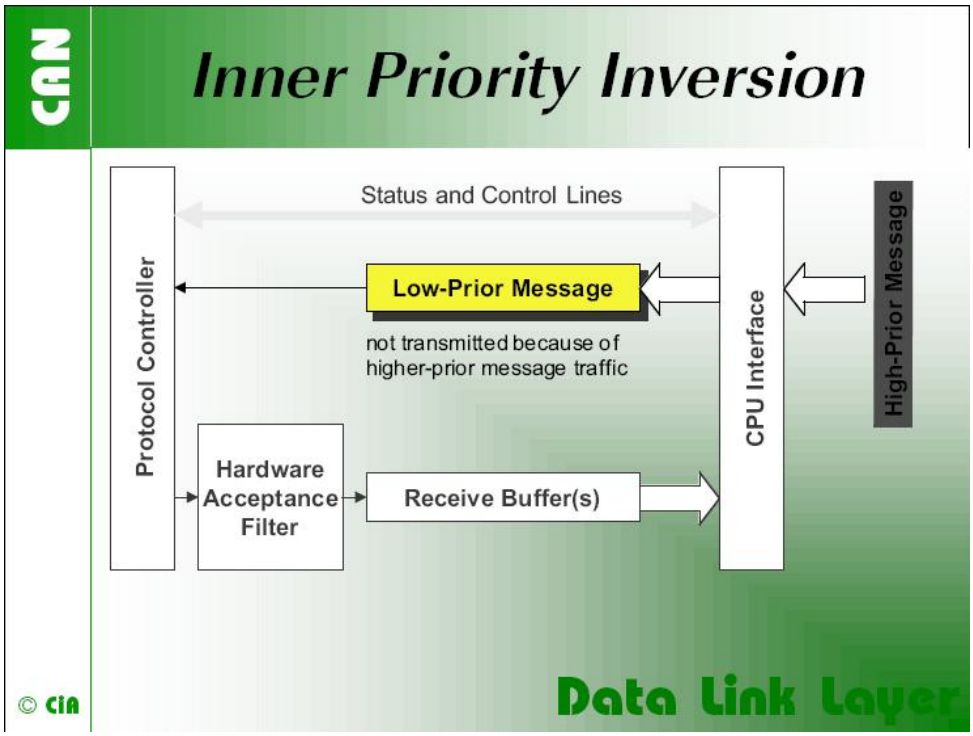
Several CAN controllers can be controlled by a single Can module as long as they belong to the same CAN Hardware Unit.

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

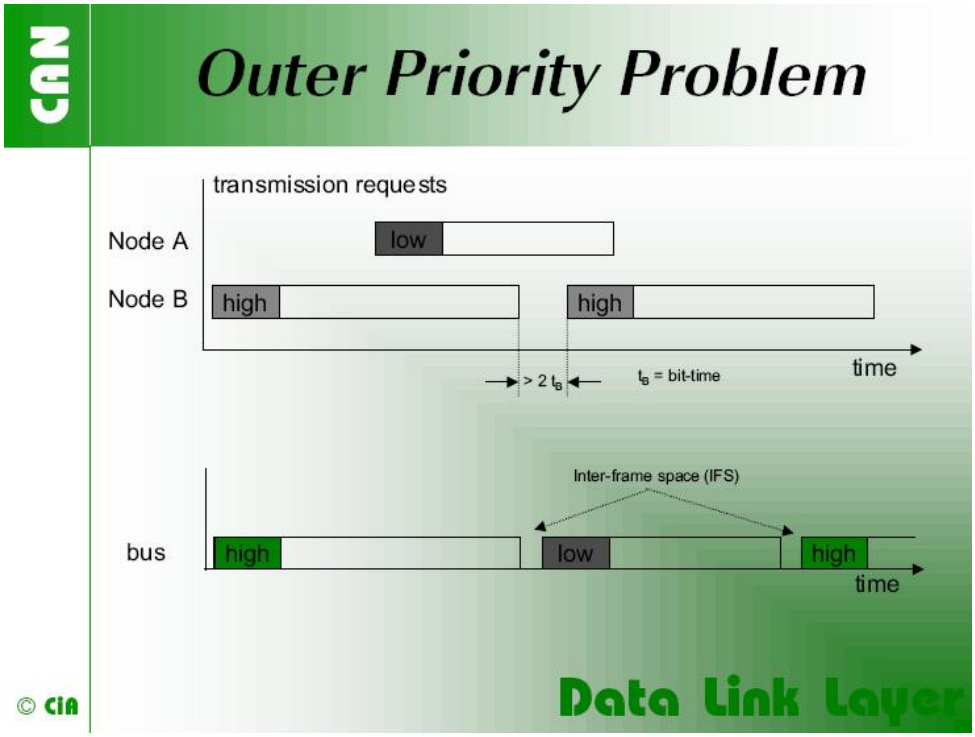
Abbreviation / Acronym:	*Description:*
CAN controller	A CAN controller serves exactly one physical channel.
CAN Hardware Unit	A CAN Hardware Unit may consists of one or multiple CAN controllers of the same type and one or multiple CAN RAM areas. The CAN Hardware Unit is either on-chip, or an external 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 [18])
CAN L-SDU	Data Link Layer Service Data Unit. Data that is transported inside the L-PDU. (see [18])
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

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)



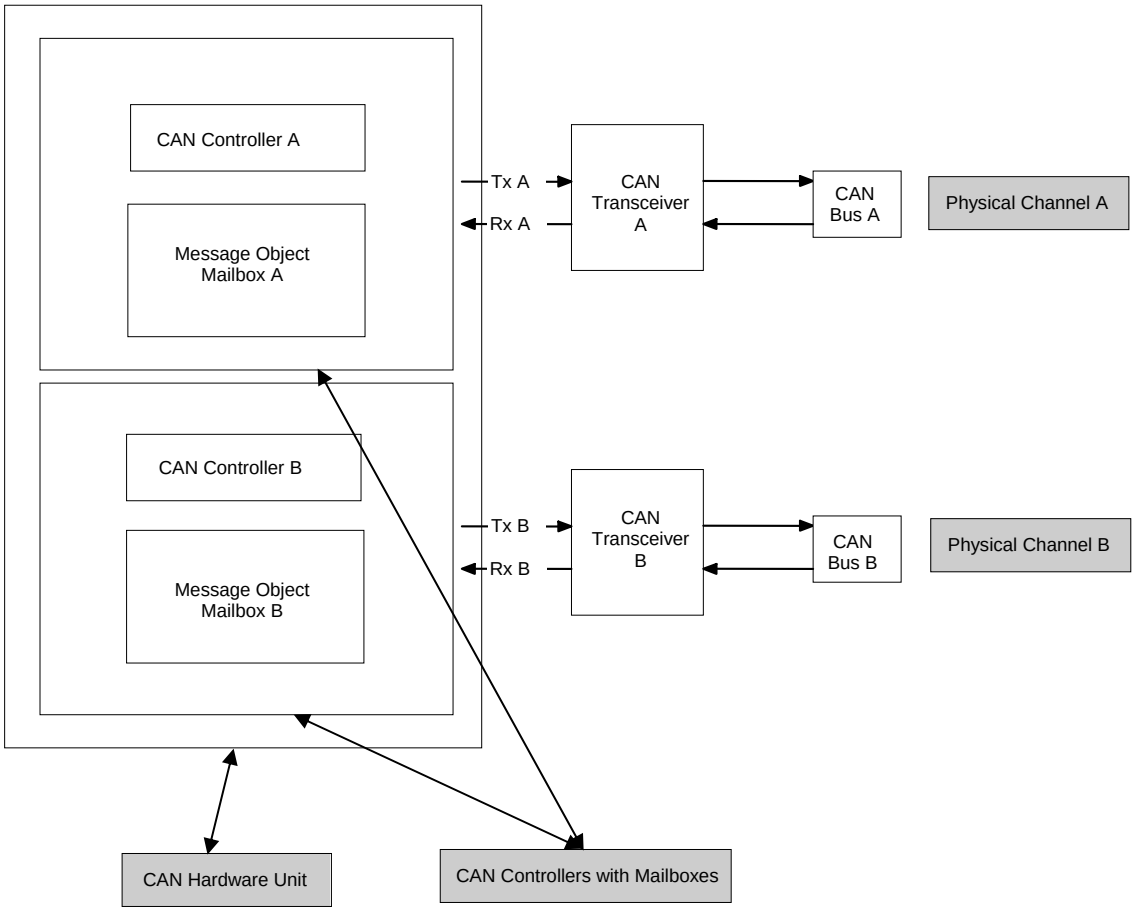
"The problem of outer priority inversion may occur in some CAN implementations. Let us assume that a CAN node wishes to transmit a package of consecutive messages with high priority, which are stored in different message buffers. If the interframe space between these messages on the CAN network is longer 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.

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

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


CAN Controller A
Message Object Mailbox A
CAN Controller B
Message Object Mailbox B
Tx A
Rx A
Tx B
Rx B
CAN Transceiver A
CAN Transceiver B
CAN Bus A
CAN Bus B
Physical Channel A
Physical Channel B
CAN Hardware Unit
CAN Controllers with Mailboxes

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

3.2 Related standards and norms

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

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

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

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.

[CAN237] ⌈The Can module shall not transmit messages triggered by remote transmission requests.⌋(BSW01147)

[CAN236] ⌈The Can module shall initialize the CAN HW to ignore any remote transmission requests.⌋(BSW01147)

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

5.1.1 Static Configuration

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

5.1.2 Driver Services

[CAN238] ⌈If the CAN controller is on-chip, the Can module shall not use any service of other drivers.⌋(BSW005)

[CAN239] ⌈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.⌋(BSW00377)

[CAN240] ⌈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.

[CAN242] ⌈If an off-chip CAN controller is used[3], the Can module shall use services of other MCAL drivers (e.g. SPI).⌋(BSW005)

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].

[CAN244] ⌈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.

[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.

5.1.3 System Services

[CAN280] 「In special hardware cases, the Can module shall poll for events of the hardware.」()

[CAN281] 「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.」()

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.

5.1.4 Can module Users

[CAN058] 「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.」(BSW12092)

5.2 File structure

5.2.1 Code file structure

[CAN078] 「The code file structure shall not be defined within this specification completely. At this point it shall be pointed out that the code-file structure shall include the following file named: Can_PBcfg.c. This file shall contain all post-build time configurable parameters.
Can_Lcfg.c is not required because the Can module does not support link-time configuration.」(BSW00380, BSW00419)

5.2.2 Header file structure

[CAN034] 「

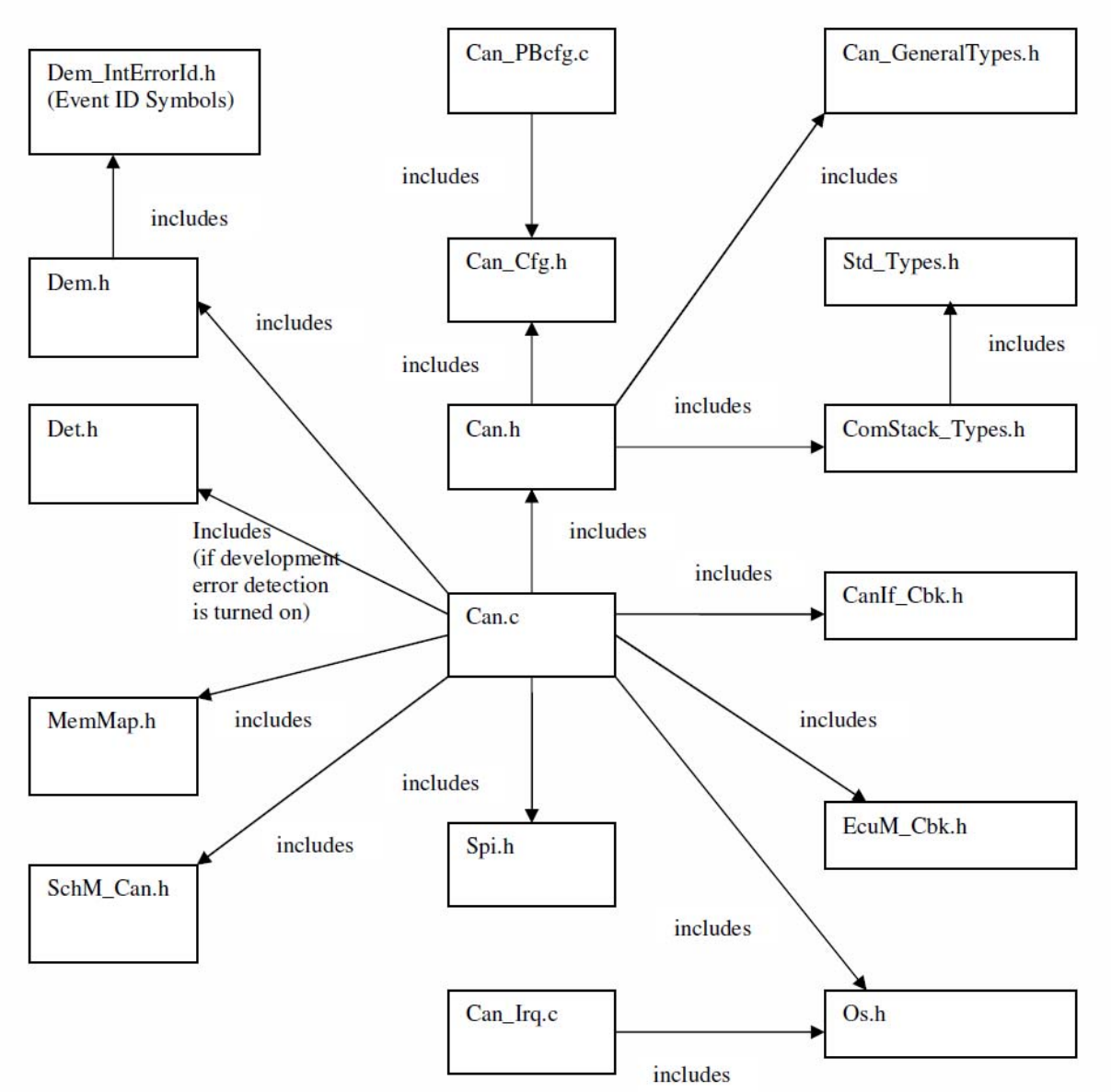


Figure 5-1: File structure for the Can module」(BSW00381, BSW00412, BSW00346, BSW158, BSW00435, BSW00436, BSW00348, BSW00301)

[CAN043]「The header file Can.h contains the declaration of the Can module API.」()

[CAN435]「The Can.h file shall include Can_GeneralTypes.h.」()

[CAN037]「The header file Can.h only contains 'extern' declarations of constants, global data, type definitions and services that are specified in the Can module SWS.」(BSW00302)

[CAN436]「Can_GeneralTypes.h shall contain all types and constants that are shared among the AUTOSAR CAN modules Can, CanIf and CanTrcv.」()

[CAN437] ⌈The integrator of the Can modules shall provide the file Can_GeneralTypes.h. ⌋()

[CAN418] ⌈Constants, global data types and functions that are only used by the Can module internally, are declared in Can.c⌋()

[CAN388] ⌈The header file Can.h shall include the header file ComStack_Types.h.⌋()

[CAN389] ⌈The implementation of the Can module shall provide the header file Can_Cfg.h that shall contain the pre-compile-time configuration parameters. ⌋ (BSW00345)

[CAN035] ⌈The file Can_Irq.c contains the implementation of interrupt frames [BSW00314]. The implementation of the interrupt service routine shall be in Can.c⌋ (BSW00314)

The Can module does not provide callback functions (no Can_Cbk.h, see also CAN244)

[CAN036] ⌈The Can module shall include the header file CanIf_Cbk.h, in which the callback functions called by the Can module at the CAN Interface module are declared.⌋(BSW00370)

[CAN390] ⌈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.⌋()

[CAN391] ⌈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.⌋()

[CAN392] ⌈If an implementation defines implementation specific production errors, the Can module shall include the header file Dem.h. By this inclusion, the APIs to report production errors as well as the required Event Id symbols are included. ⌋()

[CAN393] ⌈If the development error detection for the Can module is enabled, the Can module shall include the header file Det.h. By this inclusion, the APIs to report development errors are included.⌋()

[CAN394] 「The Can module shall include the header file MemMap.h and apply the memory mapping abstraction mechanisms as specified by [13].」(BSW00436)

[CAN397] 「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.」()

[CAN406] 「The Can module shall include the header file SchM_Can.h in order to access the module specific functionality provided by the BSW Scheduler [14].」(BSW00435)

6 Requirements traceability

Requirement	Satisfied by
-	CAN368
-	CAN420
-	CAN267
-	CAN427
-	CAN440
-	CAN404
-	CAN269
-	CAN209
-	CAN391
-	CAN215
-	CAN290
-	CAN284
-	CAN205
-	CAN268
-	CAN200
-	CAN056
-	CAN217
-	CAN280
-	CAN175
-	CAN188
-	CAN252
-	CAN397
-	CAN446
-	CAN362
-	CAN179
-	CAN180
-	CAN244
-	CAN398
-	CAN417
-	CAN373
-	CAN413
-	CAN275
-	CAN206
-	CAN386
-	CAN255

-	CAN228
-	CAN197
-	CAN390
-	CAN426
-	CAN230
-	CAN409
-	CAN190
-	CAN419
-	CAN416
-	CAN423
-	CAN183
-	CAN447
-	CAN438
-	CAN370
-	CAN439
-	CAN385
-	CAN395
-	CAN204
-	CAN187
-	CAN410
-	CAN379
-	CAN198
-	CAN227
-	CAN256
-	CAN282
-	CAN432
-	CAN174
-	CAN424
-	CAN261
-	CAN360
-	CAN229
-	CAN369
-	CAN262
-	CAN222
-	CAN444
-	CAN418
-	CAN300
-	CAN260
-	CAN226

-	CAN219
-	CAN181
-	CAN435
-	CAN281
-	CAN392
-	CAN411
-	CAN084
-	CAN442
-	CAN445
-	CAN429
-	CAN083
-	CAN283
-	CAN422
-	CAN265
-	CAN218
-	CAN405
-	CAN414
-	CAN225
-	CAN425
-	CAN408
-	CAN208
-	CAN178
-	CAN177
-	CAN199
-	CAN202
-	CAN299
-	CAN259
-	CAN189
-	CAN082
-	CAN251
-	CAN186
-	CAN363
-	CAN270
-	CAN384
-	CAN196
-	CAN441
-	CAN388
-	CAN437
-	CAN258

-	CAN080
-	CAN210
-	CAN393
-	CAN224
-	CAN412
-	CAN443
-	CAN415
-	CAN216
-	CAN264
-	CAN266
-	CAN043
-	CAN263
-	CAN434
-	CAN185
-	CAN240
-	CAN294
-	CAN436
-	CAN276
-	CAN184
-	CAN433
-	CAN361
BSW00301	CAN034
BSW00302	CAN037
BSW00306	CAN079
BSW00307	CAN999
BSW00308	CAN079
BSW00309	CAN079
BSW00312	CAN232, CAN231, CAN233, CAN214
BSW00314	CAN035
BSW00323	CAN026
BSW00325	CAN999
BSW00326	CAN999
BSW00330	CAN079
BSW00331	CAN104, CAN039
BSW00336	CAN999
BSW00337	CAN104, CAN027, CAN026, CAN028
BSW00338	CAN027, CAN028
BSW00342	CAN999
BSW00344	CAN021

BSW00345	CAN389
BSW00346	CAN034
BSW00347	CAN077
BSW00348	CAN034
BSW00353	CAN999
BSW00358	CAN223
BSW00359	CAN999
BSW00361	CAN999
BSW00369	CAN089
BSW00370	CAN036
BSW00373	CAN031
BSW00375	CAN271, CAN364
BSW00376	CAN031
BSW00377	CAN239
BSW00378	CAN999
BSW00380	CAN078
BSW00381	CAN034
BSW00383	CAN999
BSW00385	CAN104
BSW00386	CAN089
BSW00387	CAN234
BSW00395	CAN999
BSW00397	CAN999
BSW00398	CAN999
BSW00399	CAN999
BSW004	CAN111
BSW00400	CAN999
BSW00404	CAN021
BSW00405	CAN021
BSW00406	CAN103
BSW00407	CAN105
BSW00409	CAN999
BSW00412	CAN034
BSW00413	CAN999
BSW00414	CAN223
BSW00415	CAN999
BSW00417	CAN999
BSW00419	CAN078
BSW00422	CAN999

BSW00423	CAN999
BSW00424	CAN999
BSW00425	CAN999
BSW00426	CAN999
BSW00427	CAN999
BSW00428	CAN110
BSW00429	CAN999
BSW00432	CAN112, CAN108, CAN109, CAN031
BSW00433	CAN999
BSW00435	CAN406, CAN034
BSW00436	CAN394, CAN034
BSW00438	CAN291
BSW00439	CAN999
BSW00440	CAN999
BSW00442	CAN365, CAN366, CAN367
BSW00443	CAN999
BSW00444	CAN999
BSW00445	CAN999
BSW00446	CAN999
BSW00447	CAN999
BSW00449	CAN999
BSW00450	CAN431
BSW00453	CAN999
BSW00455	CAN999
BSW005	CAN242, CAN238
BSW007	CAN079
BSW01041	CAN245, CAN246
BSW01042	CAN062
BSW01043	CAN050, CAN049
BSW01045	CAN279, CAN396
BSW01049	CAN212, CAN214, CAN213
BSW01051	CAN016
BSW01053	CAN017
BSW01054	CAN271, CAN364, CAN235
BSW01055	CAN234, CAN020
BSW01059	CAN012, CAN011
BSW01060	CAN274, CAN273, CAN272
BSW01062	CAN007
BSW01122	CAN048

BSW01125	CAN999
BSW01126	CAN999
BSW01132	CAN099
BSW01133	CAN285, CAN278, CAN399, CAN400, CAN287, CAN286, CAN288
BSW01134	CAN277, CAN076, CAN401, CAN402, CAN403
BSW01135	CAN100
BSW01139	CAN062
BSW01147	CAN236, CAN237
BSW101	CAN250
BSW12056	CAN235
BSW12057	CAN245, CAN246
BSW12063	CAN060, CAN059
BSW12064	CAN999
BSW12067	CAN257
BSW12068	CAN999
BSW12069	CAN271, CAN364
BSW12075	CAN011
BSW12077	CAN372, CAN371
BSW12092	CAN058
BSW12125	CAN053
BSW12129	CAN033
BSW12163	CAN999
BSW12169	CAN017
BSW12263	CAN021
BSW12265	CAN021
BSW12448	CAN091, CAN089
BSW12461	CAN407
BSW12462	CAN999
BSW157	CAN112, CAN108, CAN109, CAN026, CAN028, CAN031
BSW158	CAN034
BSW162	CAN999
BSW164	CAN033
BSW167	CAN023
BSW168	CAN999
BSW170	CAN999

Document: General requirements on Basic Software [2]

Requirement	***Satisfied by***
[BSW00344] Reference to link-time configuration	CAN021

[BSW00404] Reference to post build time configuration	CAN021
[BSW00405] Reference to multiple configuration sets	CAN021
[BSW00345] Pre-Build Configuration	CAN389
[BSW159] Tool-based configuration	CAN022
[BSW167] Static configuration checking	CAN023, CAN024
[BSW171] Configurability of optional functionality	CAN064_Conf, CAN095_Conf, CAN069_Conf
[BSW170] Data for reconfiguration of SW-components	not applicable (doesn't concern this document)
[BSW00380] C-Files for configuration parameters	CAN078
[BSW00419] Separate C-Files for pre-compile time configuration	CAN078
[BSW00381] Separate configuration header file for pre-compile time parameters	CAN034
[BSW00412] Separate H-File for configuration parameters	CAN034
[BSW00383] List dependencies of configuration files	not applicable (implementation specific documentation)
[BSW00384] List dependencies to other modules	Chapter 5
[BSW00387] Specify the configuration class of callback function	CAN234
[BSW00388] Introduce containers	Chapter 10.2
[BSW00389] Containers shall have names	Chapter 10.2
[BSW00390] Parameter content shall be unique within the module	Chapter 10.2
[BSW00391] Parameter shall have unique names	Chapter 10.2
[BSW00392] Parameters shall have a type	Chapter 10.2
[BSW00393] Parameters shall have a range	Chapter 10.2
[BSW00394] Specify the scope of the parameters	Chapter 10.2
[BSW00395] 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)
[BSW00396] Configuration classes	Chapter 10.2
[BSW00397] Pre-compile-time parameters	Not applicable: this is not a requirement but a definition of term.
[BSW00398] Link-time parameters	Not applicable: this is not a requirement but a definition of term.
[BSW00399] Loadable Post-build time parameters	Not applicable: this is not a requirement but a definition of term.
[BSW00400] Selectable Post-build time parameters	Not applicable: this is not a requirement but a definition of term.
[BSW00438] Post Build Configuration Data Structure	CAN291
[BSW00402] Published information	Chapter 10.2.2
[BSW00375] Notification of wake-up reason	CAN271, CAN364
[BSW101] Initialization interface	CAN250
[BSW168] Diagnostic Interface of SW components	not applicable (requirement for the diagnostic services, not for the BSW module)
[BSW00416] Sequence of Initialization	not applicable (this is a general software integration requirement)
[BSW00406] Check module initialization	CAN103, defined development error CAN_E_UNINIT
[BSW00437] NoInit--Area in RAM	not applicable

[BSW00407] Function to read out published parameters	CAN105, CAN106_Conf
[BSW00423] Usage of SW-C template to describe BSW modules with AUTOSAR Interfaces	not applicable (this module does not provide an AUTOSAR interface)
[BSW00424] BSW main processing function task allocation	not applicable (requirement on system design, not on a single module)
[BSW00425] Trigger conditions for schedulable objects	not applicable (trigger conditions are system configuration specific.)
[BSW00426] Exclusive areas in BSW modules	not applicable (no exclusive areas defined)
[BSW00427] ISR description for BSW modules	not applicable (no ISR's defined for this module, usage of interrupts is implementation specific)
[BSW00428] Execution order dependencies of main processing functions	CAN110
[BSW00429] Restricted BSW OS functionality access	not applicable (requirement on the implementation, not for the specification)
[BSW00432] Modules should have separate main processing functions for read/receive and write/transmit data path	CAN031, CAN108, CAN109, CAN112
[BSW00433] Calling of main processing functions	not applicable (requirement on system design, not on a single module)
[BSW00450] Main Function Processing for Un-Initialized Modules	CAN431
[BSW00442] Debugging Support in Modules	CAN365, CAN366, CAN367
[BSW00336] Shutdown interface	not applicable
[BSW00337] Classification of errors	CAN026, CAN027, CAN028, CAN104
[BSW00338] Detection and Reporting of development errors	CAN028, CAN027
[BSW00369] Do not return development error codes via API	CAN089
[BSW00339] Reporting of production relevant errors and exceptions	CAN113_Conf
[BSW00422] Debouncing of production relevant error status	not applicable (requirement on the DEM)
[BSW00417] Reporting of Error Events by Non-Basic Software	not applicable (this is a BSW module)
[BSW00323] API parameter checking	CAN026
[BSW004] Version check	CAN111
[BSW00409] Header files for production code error IDs	not applicable (no production errors codes used by Can module)
[BSW00385] List possible error notifications	CAN104
[BSW00386] Configuration for detecting an error	CAN089
[BSW00455] Implementation Conformance Class 1 and 2 (ICC1 and ICC2) Guidelines	not applicable
[BSW161] Microcontroller abstraction	Chapter 1
[BSW162] ECU layout abstraction	not applicable (done in CanIf module)
[BSW005] No hard coded horizontal interfaces within MCAL	CAN238, CAN242
[BSW00415] User dependent include files	not applicable (only one user for this module)
[BSW164] Implementation of interrupt service routines	CAN033

[BSW00325] Runtime of interrupt service routines	not applicable (The runtime is not under control of the Can module, because callback functions are called.)
[BSW00326] 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
[BSW00342] Usage of source code and object code	not applicable (Only source code delivery is supported)
[BSW00343] Specification and configuration of time	CAN113_Conf, CAN355_Conf, CAN356_Conf, CAN357_Conf, CAN358_Conf, CAN376_Conf
[BSW160] Human-readable configuration data	CAN047
[BSW00453] Harmonization of BSW Modules	not applicable, yet
[BSW007] HIS MISRA C	CAN079
[BSW00300] Module naming convention	is fulfilled, see function definitions in 8.3
[BSW00413] Accessing instances of BSW modules	not applicable (this requirement is fulfilled by the CanIf module specification)
[BSW00347] Naming separation of drivers	CAN077
[BSW00441] Enumeration literals and #define naming convention	Chapter 8.2.6, Chapter 8.2.7
[BSW00305] Self-defined data types naming convention	is fulfilled, see type definitions in 8.2
[BSW00307] Global variables naming convention	not applicable (because no global variables are specified for Can module)
[BSW00310] API naming convention	is fulfilled, see function definitions in 8.3
[BSW00373] Main processing function naming convention	CAN031
[BSW00327] Error values naming convention	Chapter 7.10.1 error names have been selected accordingly
[BSW00335] Status values naming convention	Chapter 7.2 is fulfilled by state description
[BSW00350] Development error detection keyword	CAN064_Conf
[BSW00408] Configuration parameter naming convention	Chapter 10.2
[BSW00410] Compiler switches shall have defined values	Chapter 10.2
[BSW00411] Get version info keyword	CAN106_Conf
[BSW00346] Basic set of module files	CAN034
[BSW158] Separation of configuration from implementation	CAN034
[BSW00314] Separation of interrupt frames and service routines	CAN035
[BSW00370] Separation of callback interface from API	CAN036
[BSW00435] Module Header File Structure for the Basic Software Scheduler	CAN034, CAN406
[BSW00436] Module Header File Structure for the Basic Software Memory Mapping	CAN034, CAN394
[BSW00447] Standardizing Include file structure of BSW Modules Implementing Autosar Service	not applicable
[BSW00348] Standard type header	CAN034
[BSW00353] Platform specific type header	not applicable (automatically included with Standard types)
[BSW00361] Compiler specific language extension header	not applicable
[BSW00301] Limit imported information	CAN034
[BSW00302] Limit exported information	CAN037

[BSW00328] Avoid duplication of code	Implementation requirement Fulfilled e.g. by defining one Can module that controls multiple channels
[BSW00312] Shared code shall be reentrant	CAN214, CAN231, CAN232, CAN233
[BSW006] Platform independency	Chapter 1
[BSW00439] Declaration of interrupt handlers and ISRs	not applicable
[BSW00448] Module SWS shall not contain requirements from Other Modules	All chapters of this document containing SWS items
[BSW00449] BSW Service APIs used by Autosar Application Software shall return a Std_ReturnType	not applicable
[BSW00357] Standard API return type	not used
[BSW00377] Module Specific API return type	CAN039
[BSW00304] AUTOSAR integer data types	standard integer data types are used
[BSW00355] Do not redefine AUTOSAR integer data types	no redefined integer types in 8.2
[BSW00378] AUTOSAR boolean type	not applicable (not used)
[BSW00306] Avoid direct use of compiler and platform specific keywords	CAN079
[BSW00308] Definition of global data	CAN079
[BSW00309] Global data with read-only constraint	CAN079
[BSW00371] Do not pass function pointers via API	Chapter 8.3 (function definitions)
[BSW00358] Return type of init() functions	CAN223
[BSW00414] Parameter of init function	CAN223
[BSW00376] Return type and parameters of main processing functions	CAN031
[BSW00359] Return type of callback functions	not applicable (no callback functions implemented in Can module)
[BSW00360] Parameters of callback functions	no callbacks implemented in Can module
[BSW00440] Function prototype for callback functions of AUTOSAR Services	not applicable
[BSW00329] Avoidance of generic interfaces	No generic interface used. Still content of functions might be configuration dependent. Scope of function is always defined
[BSW00330] Usage of macros instead of functions	CAN079
[BSW00331] Separation of error and status values	CAN104, CAN039
[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
[BSW009], [BSW00401], [BSW172], [BSW010], [BSW00333], [BSW00374], [BSW00379], [BSW003], [BSW00318], [BSW00321], [BSW00341], [BSW00334]	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***

[BSW12263] Object code compatible configuration concept	CAN021
[BSW12056] Configuration of notification mechanisms	CAN235
[BSW12267] Configuration of wake-up sources	CAN330_Conf
[BSW12057] Driver module initialization	CAN245, CAN246
[BSW12125] Initialization of hardware resources	CAN053
[BSW12163] Driver module de-initialization	not applicable (decision in JointMM Meeting: no de-initialization for drivers that don't need to store non volatile information)
[BSW12461] Responsibility for register initialization	CAN407
[BSW12462] Provide settings for register initialization	not applicable (Software Documentation Requirements are not covered in the CAN Driver SWS)
[BSW12463] Combine and forward settings for register initialization	CAN024
[BSW12068] MCAL initialization sequence	not applicable (requirement on ECU state manager)
[BSW12069] Wake-up notification of ECU State Manager	CAN271, CAN364
[BSW157] Notification mechanisms of drivers and handlers	CAN026, CAN028, CAN031, CAN108, CAN109, CAN112
[BSW12169] Control of operation mode	CAN017
[BSW12063] Raw value mode	CAN059, CAN060
[BSW12075] Use of application buffers	CAN011
[BSW12129] Resetting of interrupt flags	CAN033
[BSW12064] Change of operation mode during running operation	not applicable
[BSW12448] Behavior after development error detection	CAN091, CAN089
[BSW12067] Setting of wake-up conditions	CAN257
[BSW12077] Non-blocking implementation	CAN371, CAN372
[BSW12078] Runtime and memory efficiency	no effect on API definition implementation requirement
[BSW12092] Access to drivers	CAN058
[BSW12265] Configuration data shall be kept constant	CAN021 (stored in ROM -> implicitly constant)
[BSW12264] Specification of configuration items	Chapter 10

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

Requirement	***Satisfied by***
[BSW01125] Data throughput read direction	not applicable (requirement affects complete COM stack and will not be broken down for the individual layers)
[BSW01126] Data throughput write direction	not applicable (requirement affects complete COM stack and will not be broken down for the individual layers)
[BSW01139] CAN controller specific initialization	CAN062
[BSW01033] Basic Software Modules Requirements	see table above
[BSW01034] Hardware independent implementation	Chapter 1
[BSW01035] Multiple CAN controller support	Chapter 1
[BSW01036] CAN Identifier Length Configuration	CAN065_Conf
[BSW01037] Hardware Filter Configuration	CAN066_Conf, CAN325_Conf

[BSW01038] Bit Timing Configuration	CAN005_Conf, CAN073_Conf, CAN074_Conf, CAN075_Conf
[BSW01039] CAN Hardware Object Handle definitions	CAN324_Conf
[BSW01040] HW Transmit Cancellation configuration	CAN069_Conf
[BSW01058] Configuration of multiplexed transmission	CAN095_Conf
[BSW01062] Configuration of polling mode	CAN007, CAN314_Conf, CAN317_Conf, CAN318_Conf, CAN319_Conf,
[BSW01135] Configuration of multiple TX Hardware Objects	CAN100
[BSW01041] Can module Module Initialization	CAN245, CAN246
[BSW01042] Selection of static configuration sets	CAN062
[BSW01043] Enable/disable Interrupts	CAN049, CAN050
[BSW01059] Data Consistency	CAN011, CAN012
[BSW01045] Reception Indication Service	CAN279, CAN396
[BSW01049] Dynamic transmission request service	CAN212, CAN213, CAN214
[BSW01051] Transmit Confirmation	CAN016
[BSW01053] CAN controller mode select	CAN017
[BSW01054] Wake-up Notification	CAN235, CAN271, CAN364
[BSW01132] Mixed mode for notification detection on CAN HW	CAN099
[BSW01133] HW Transmit Cancellation Support	CAN285, CAN286, CAN287, CAN288, CAN278, CAN399, CAN400
[BSW01134] Multiplexed Transmission	CAN076, CAN277, CAN401, CAN402, CAN403
[BSW01055] Bus-off Notification	CAN020, CAN234
[BSW01060] no automatic bus-off recovery	CAN272, CAN273, CAN274
[BSW01122] Support for wakeup during sleep transition	CAN048
[BSW01147] No Remote Frame Support	CAN236, CAN237

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]).

[CAN033] 「The Can module shall implement the interrupt service routines for all CAN Hardware Unit interrupts that are needed. 」(BSW164, BSW12129)

[CAN419] 「The Can module shall disable all unused interrupts in the CAN controller. 」()

[CAN420] 「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.

[CAN079] 「The Can module shall fulfill all design and implementation guidelines described in [11].」(BSW007, BSW00306, BSW00308, BSW00309, BSW00330)

7.1 Driver scope

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

[CAN077] 「For CAN Hardware Units of different type, different Can modules shall be implemented. 」(BSW00347)

[CAN284] 「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>()`
BSW00347 specifies the naming convention.

[CAN385] 「The naming conventions shall be used only in that case, if multiple different CAN controller types on one ECU have to be supported. 」()

[CAN386] 「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.

[CAN103] 「After power-up/reset, the Can module shall be in the state CAN_UNINIT. 」(BSW00406)

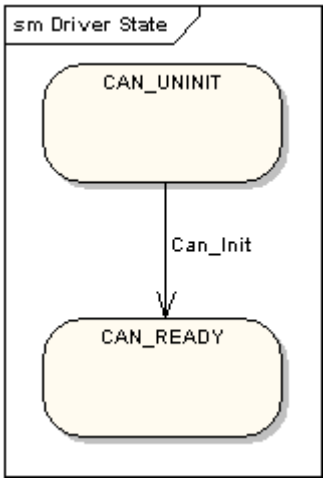


Figure 7-1

[CAN246] 「The function Can_Init shall change the module state to CAN_READY, after initializing all controllers inside the HW Unit.」(BSW12057, BSW01041)

[CAN245]「The function Can_Init shall initialize all CAN controllers according to their configuration.」(BSW12057, BSW01041)

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_ChangeBaudrate 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).

[CAN257] 「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.」(BSW12067)

[CAN258] 「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.」()

[CAN404] 「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.

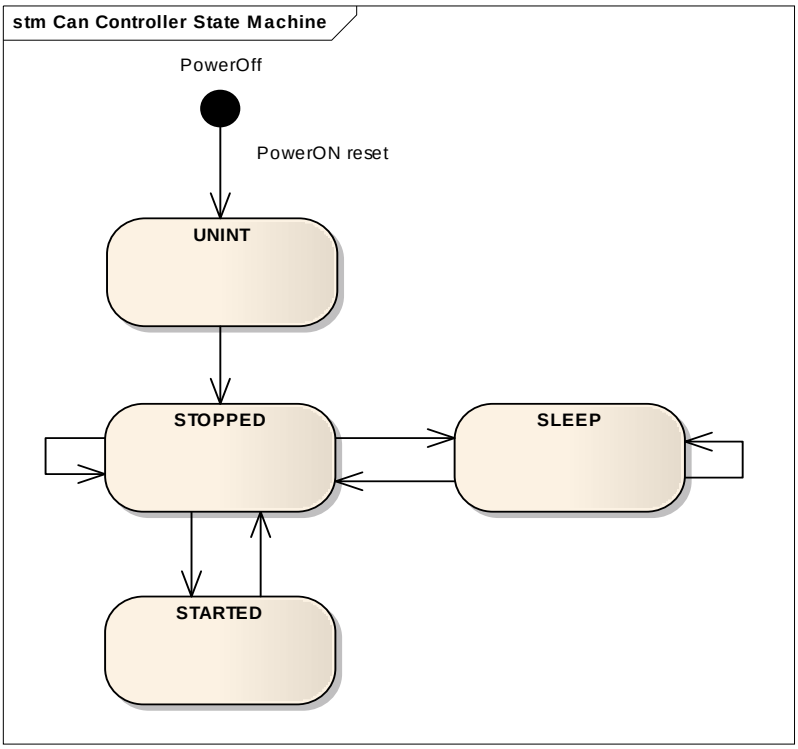


Figure 7-2

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.

[CAN259] ⌈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 CAN174 and CAN408).

7.3.4 State transition caused by function Can_ChangeBaudrate

- STOPPED → STOPPED
- software triggered by the function call Can_ChangeBaudrate
- changes the CAN controller configuration

CAN controller registers are set according to the static configurations.

[CAN256] ⌈The Can module's environment shall only call Can_ChangeBaudrate when the CAN controller is in state STOPPED.⌋()

[CAN260] ⌈The function Can_ChangeBaudrate shall maintain the CAN controller in the state STOPPED.⌋()

[CAN422] ⌈The function Can_ChangeBaudrate shall ensure that any settings that will cause the CAN controller to participate in the network are not set.⌋()

When the function Can_ChangeBaudrate is entered and the CAN controller is not in state STOPPED, it shall raise the error CAN_E_TRANSITION (Compare to CAN190).

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.

[CAN370] ⌈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.⌋()

[CAN371] ⌈This polling shall take the maximum time of `CanTimeoutDuration` for blocking function and thus the polling time is limited.⌋(BSW12077)

[CAN398] ⌈The function Can_SetControllerMode shall use the system service GetCounterValue for timeout monitoring to avoid blocking functions.⌋()

[CAN372] ⌈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.⌋ (BSW12077)

[CAN373] ⌈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

[CAN261] ⌈The function Can_SetControllerMode(CAN_T_START) shall set the hardware registers in a way that makes the CAN controller participating on the network.⌋()

[CAN262] ⌈The function Can_SetControllerMode(CAN_T_START) shall wait for limited time until the CAN controller is fully operational. Compare to CAN371.⌋()

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.

[CAN409] ⌈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 CAN200).⌋()

State transition caused by function Can_SetControllerMode(CAN_T_STOP)

- STARTED → STOPPED
- software triggered

[CAN263] ⌈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.⌋()

[CAN264] ⌈The function Can_SetControllerMode(CAN_T_STOP) shall wait for a limited time until the CAN controller is really switched off. Compare to CAN371.⌋()

[CAN282] ⌈ The function Can_SetControllerMode(CAN_T_STOP) shall cancel pending messages. ⌋()

[CAN283] ⌈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.

[CAN410] ⌈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 CAN200).⌋()

State transition caused by function Can_SetControllerMode(CAN_T_SLEEP)

- STOPPED → SLEEP
- software triggered

[CAN265] ⌈ The function Can_SetControllerMode(CAN_T_SLEEP) shall set the controller into sleep mode.⌋()

[CAN266] ⌈ 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 CAN371.」()

[CAN290] 「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.」()

[CAN405] 「This logical sleep mode shall left only, if function Can_SetControllerMode(CAN_T_WAKEUP) is called.」()

[CAN411] 「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 CAN200).」()

State transition caused by function Can_SetControllerMode(CAN_T_WAKEUP)

- SLEEP → STOPPED
- software triggered

[CAN267] 「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).」()

[CAN268] 「The function Can_SetControllerMode(CAN_T_WAKEUP) shall wait for a limited time until the CAN controller is in STOPPED state. Compare to CAN371.」()

[CAN412] 「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 CAN200).」()

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.

[CAN270]「On hardware wakeup (triggered by a wake-up event from CAN bus), the CAN controller shall transition into the state STOPPED.」()

[CAN271]「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.」(BSW00375, BSW12069, BSW01054)

[CAN269]「The Can module shall not further process the L-PDU that caused a wake-up.」()

[CAN048]「In case of a CAN bus wake-up during sleep transition, the function Can_SetControllerMode(CAN_T_WAKEUP) shall return CAN_NOT_OK.」(BSW01122)

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

[CAN020]「
- 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.」(BSW01055)

[CAN272]「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.」(BSW01060)

[CAN273]「After bus-off detection, the Can module shall cancel still pending messages without raising a cancellation notification.」(BSW01060)

[CAN274]「The Can module shall disable or suppress automatic bus-off recovery.」(BSW01060)

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.

[CAN250]「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」(BSW101)

[CAN053] 「Can_Init shall not change registers of CAN controller Hardware resources that are not used. 」(BSW12125)

The Can module shall apply the following rules regarding initialization of controller registers:

- **[CAN407]** 「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.」(BSW12461)

[CAN056] 「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. 」()

[CAN023] 「The consistency of the configuration must be checked by the configuration tool(s).」(BSW167)

[CAN062] 「The function Can_ChangeBaudrate shall re-initialize the CAN controller and the controller specific settings.」(BSW01139, BSW01042)

The CanIf module must first set the CAN controller in STOPPED state. Then Can_ChangeBaudrate can be invoked by the appropriate upper layer.

[CAN255] 「The function Can_ChangeBaudrate shall only affect register areas that contain specific configuration for a single CAN controller. 」()

[CAN021] 「The desired CAN controller configuration can be selected with the parameter Config. 」(BSW00344, BSW00404, BSW00405, BSW12263, BSW12265)

[CAN291] 「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.」(BSW00438)

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.

[CAN059] ⌈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.⌋(BSW12063)

[CAN427] ⌈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.⌋()

[CAN100] ⌈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.⌋(BSW01135)

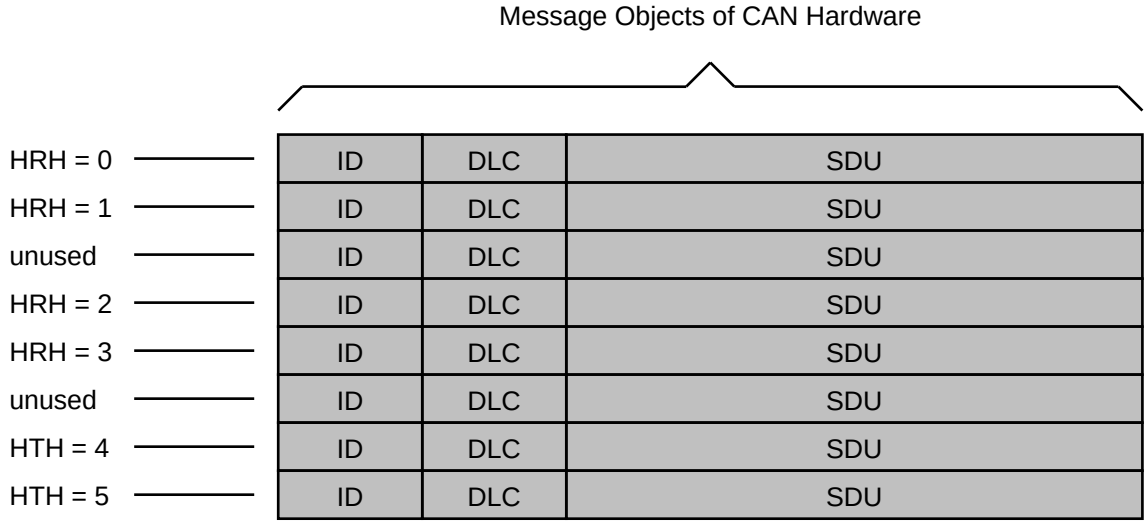


Figure 7-3: Example of assignment of HTHs and HRHs to the Hardware Objects. The numbering of HTHs and HRHs are implementation specific. The chosen numbering is only an example.

[CAN276]「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 CAN276 is used to reduce time for searching in the CanIf module implementation.

[CAN016]「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.」(BSW01051)

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

[CAN277]「The Can module shall allow that the functionality "Multiplexed Transmission" is statically configurable (ON | OFF) at pre-compile time.」(BSW01134)

[CAN401]「Several transmit hardware objects shall be assigned by one HTH to represent one transmit entity to the upper layer.」(BSW01134)

[CAN402]「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.」(BSW01134)

[CAN403]「The Can module shall support multiplexed transmission for devices, which send L-PDUs in order of L-PDU priority.」(BSW01134)

[CAN076]「The Can module shall NOT support software emulation for the transmission in order of LPDU-priority.」(BSW01134)

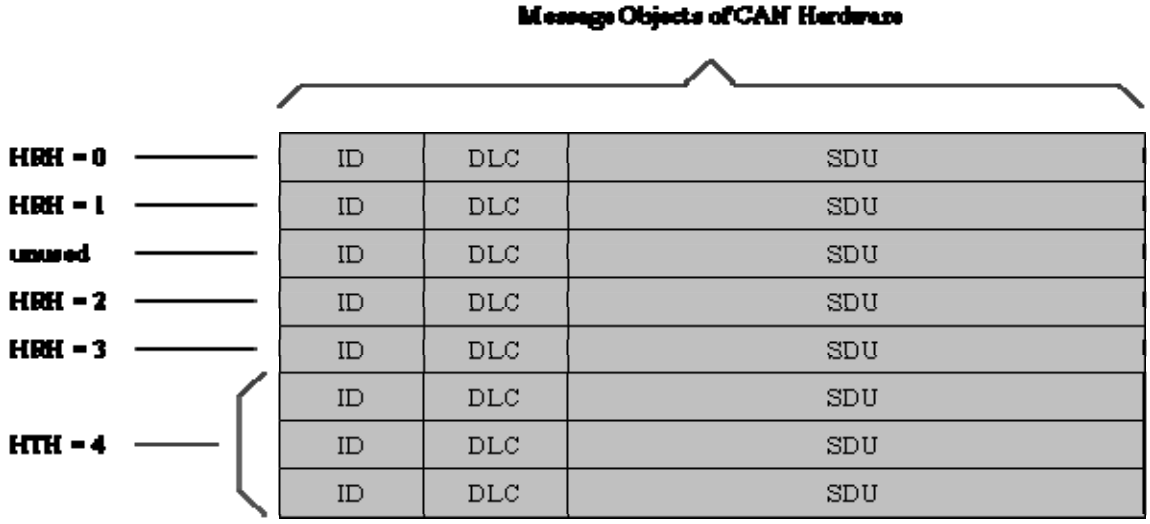


Figure 7-4: Example of assignment of HTHs and HRHs to the Hardware Objects with multiplexed transmission. The numbering of HTHs and HRHs are implementation specific. The chosen numbering is only an example.

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.

[CAN278] 「The Can module shall allow that the functionality “Transmit Cancellation” is statically configurable (ON | OFF) at pre-compile time.」(BSW01133)

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

[CAN432] 「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.*」()

[CAN285] 「Transmit cancellation may only be used when transmit buffers are enabled inside the CanIf module.」(BSW01133)

[CAN286] ⌈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.⌋(BSW01133)

[CAN433] ⌈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:

[CAN399] ⌈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.⌋(BSW01133)

[CAN400] ⌈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.⌋(BSW01133)

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

[CAN287] ⌈The Can module shall raise a notification when the cancellation was successful by calling the function CanIf_CancelTxConfirmation.⌋(BSW01133)

[CAN288] ⌈The TX request for the new L-PDU shall be repeated by the CanIf module, inside the notification function CanIf_CancelTxConfirmation. ⌋(BSW01133)

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

[CAN011] ⌈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).⌋(BSW12075, BSW01059)

7.6 L-PDU reception

[CAN279] ⌈On L-PDU reception, the Can module shall call the RX indication callback function CanIf_RxIndication with ID, DLC and pointer to the L-SDU buffer as parameter.⌋(BSW01045)

[CAN423] ⌈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.⌋()

[CAN396] ⌈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.⌋(BSW01045)

[CAN060] ⌈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.⌋ (BSW12063)

7.6.1 Receive Data Consistency

[CAN299] ⌈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.⌋()

[CAN300] ⌈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.

[CAN012] ⌈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.⌋(BSW01059)

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.

[CAN395] 「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.

[CAN364] 「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.」 (BSW00375, BSW12069, BSW01054)

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 Interface will then inform the ECU State Manager of the wakeup event.

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.

[CAN099] ⌈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. ⌋(BSW01132)

[CAN007] ⌈It shall be possible to configure the driver such that no interrupts at all are used (complete polling). ⌋(BSW01062)

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:

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

[CAN104] ⌈The Can module shall be able to detect the following errors and exceptions depending on its configuration (development/production)⌋(BSW00337, BSW00385, BSW00331)

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

7.10.1 Development Errors

[CAN026] ⌈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. ⌋(BSW00337, BSW00323, BSW157)

[CAN028] ⌈The Can module shall call the Development Error Tracer when DET is switched on and the Can module detects an error. ⌋(BSW00337, BSW00338, BSW157)

[CAN091] ⌈After return of the DET the Can module's function that raised the development error shall return immediately.⌋(BSW12448)

[CAN089] ⌈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. ⌋(BSW00369, BSW00386, BSW12448)

[CAN080] ⌈Development error values are of type uint8.⌋()

7.10.2 Production Errors

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

7.10.3 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.

7.11 Error detection

[CAN082] ⌈The detection of development errors is configurable (*ON* / *OFF*) at pre-compile time. The switch CanDevErrorDetection (see chapter 10) shall activate or deactivate the detection of all development errors.⌋()

[CAN083] ⌈If the CanDevErrorDetection switch is enabled API parameter checking is enabled. The detailed description of the detected errors can be found in chapter 7.10. ⌋()

[CAN084] ⌈The detection of production code errors cannot be switched off.⌋()

7.12 Error notification

[CAN027] ⌈ Detected development errors shall be reported to the `Det_ReportError` service of the Development Error Tracer (DET) if the pre-processor switch `CanDevErrorDetection` is set (see chapter 10).⌋(BSW00337, BSW00338)

[CAN424] ⌈No code for catching development errors shall be generated, when development errors are switched off. ⌋()

7.13 Version Check

[CAN111] ⌈The CAN module shall perform Inter Module Checks to avoid integration of incompatible files.

The imported included files shall be checked by preprocessing directives. 」(BSW004)

The following version numbers shall be verified:
- <MAB>_AR_RELEASE_MAJOR_VERSION
- <MAB>_AR_RELEASE_MINOR_VERSION
Where <MAB> is the Module Abbreviation of the other (external) modules which provide header files included by the CAN module.

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

7.14 Debugging

[CAN365] 「Each variable that shall be accessible by AUTOSAR Debugging, shall be defined as global variable.」(BSW00442)

[CAN366] 「All type definitions of variables which shall be debugged, shall be accessible by the header file Can.h.」(BSW00442)

[CAN367] 「The declaration of variables in the header file shall be such, that it is possible to calculate the size of the variables by C-"sizeof" operation.」(BSW00442)

8 API specification

The prefix of the function names may be changed in an implementation with several Can modules as described in CAN284.

8.1 Imported types

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

[CAN222] ⌈

Module	*Imported Type*
CanIf	CanIf_ControllerModeType
ComStack_Types	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

[CAN438]⌈The content of Can_GeneralTypes.h consists of types specified within [5] and the following type specifications within this document except Can_ConfigType. ⌋ ()

[CAN439]⌈The content of Can_GeneralTypes.h shall be protected by a CAN_GENERAL_TYPES define. ⌋()

[CAN440]⌈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

[CAN413] ⌈

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_ControllerBaudrateConfigType

[CAN414] ⌈

Name:	`Can_ControllerBaudrateConfigType`
Type:	`Structure`
Range:	Implementation specific.
Description:	This is the type of the external data structure containing the bit timing related initialization data for one CAN controller. The contents of the initialization data structure are CAN hardware specific.

⌋()

8.2.3 Can_PduType

[CAN415] ⌈

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.		

⌋()

8.2.4 Can_IdType

[CAN416] ⌈

Name:	`Can_IdType`		
Type:	`uint16, uint32`		
Range:	`Standard`	`--`	0..0x7FF
	`Extended`	`--`	0..0xFFFFFFFF
Description:	Represents the Identifier of an L-PDU. For extended IDs the most significant bit is set.		

」()

8.2.5 Can_HwHandleType

[CAN429] 「

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.		

」()

8.2.6 Can_StateTransitionType

[CAN417] 「

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

[CAN039] 「

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 .	

」(BSW00331)

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

[CAN223] ⌈

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.	

⌋(BSW00358, BSW00414)

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

[CAN174] ⌈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.⌋()

[CAN408] ⌈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.⌋()

[CAN175] ⌈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

[CAN224] ⌈

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.

」()

[CAN105] 「The function Can_GetVersionInfo shall return the version information of this module. The version information includes:

- Module Id
- Vendor Id
- Vendor specific version numbers (BSW00407).」(BSW00407)

[CAN251] 「 If source code for caller and callee is available, the function Can_GetVersionInfo should be realized as a macro, defined in the Can module's header file.」()

[CAN177] 「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.」()

[CAN252] 「The function Can_GetVersionInfo shall be pre compile time configurable (`On/Off`) by the configuration parameter: `CanVersionInfoApi`.」()

8.3.1.3 Can_CheckBaudrate

[CAN454] 「

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	

」()

[CAN455] 「The service `Can_CheckBaudrate(Controller, Baudrate)` shall be called by `CanIf_CheckBaudrate()` for the requested CAN controller. 」()

[CAN456] 「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.」()

[CAN457] 「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.」()

[CAN458] 「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.」()

[CAN459] 「Caveats of `Can_CheckBaudrate(Controller, Baudrate)`:

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

[CAN460] 「Configuration of `Can_CheckBaudrate(Controller, Baudrate)`: If Can supports changing of the baudrate and thus this service, shall be configurable via `CAN_CHANGE_BAUDRATE_SUPPORT.」()`

8.3.2 Services affecting one single CAN Controller

8.3.2.1 Can_ChangeBaudrate

[CAN449] ⌈

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.	

⌋()

The function Can_ChangeBaudrate re-initializes the CAN controller and the controller specific settings (see CAN062).

Different sets of static configuration may have been configured. The parameter *Config points to the hardware specific structure that describes the configuration (see CAN291).

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 CAN256 and CAN260).

The CAN controller is in state STOPPED after (re-)initialization (see CAN259).

[CAN450] ⌈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.⌋()

[CAN451] ⌈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.⌋()

[CAN452] ⌈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.⌋()

[CAN453] ⌈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.⌋()

[CAN461] ⌈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_SetControllerMode

[CAN230] ⌈

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.	

⌋()

[CAN017] ⌈The function Can_SetControllerMode shall perform software triggered state transitions of the CAN controller State machine. See also [BSW12169]⌋ (BSW12169, BSW01053)

[CAN384] ⌈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_ChangeBaudrate or Can_Init.⌋()

Refer to CAN048 for the case of a wakeup event from CAN bus occurred during sleep transition.

[CAN294] ⌈The function Can_SetControllerMode shall disable the wake-up interrupt, while checking the wake-up status. ⌋()

[CAN196] ⌈The function Can_SetControllerMode shall enable interrupts that are needed in the new state. ⌋()

[CAN425] ⌈Enabling of CAN interrupts shall not be executed, when CAN interrupts have been disabled by function Can_DisableControllerInterrupts.⌋()

[CAN197] ⌈The function Can_SetControllerMode shall disable interrupts that are not allowed in the new state. ⌋()

[CAN426] ⌈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.

[CAN198] ⌈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.⌋()

[CAN199] ⌈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.⌋()

[CAN200] ⌈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.⌋()

8.3.2.3 Can_DisableControllerInterrupts

[CAN231] ⌈

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.	

⌋(BSW00312)

[CAN049] ⌈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.⌋(BSW01043)

[CAN202] ⌈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).

[CAN204] ⌈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.

[CAN205] ⌈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.⌋()

[CAN206] ⌈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.⌋()

8.3.2.4 Can_EnableControllerInterrupts

[CAN232] ⌈

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.	

⌋(BSW00312)

[CAN050] ⌈The function Can_EnableControllerInterrupts shall enable all interrupts that must be enabled according the current software status.⌋(BSW01043)

CAN202 applies to this function.

[CAN208] ⌈The function Can_EnableControllerInterrupts shall perform no action when Can_DisableControllerInterrupts has not been called before.⌋()

See also implementation example for Can_DisableControllerInterrupts.

[CAN209] ⌈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.⌋()

[CAN210] ⌈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.⌋()

8.3.2.5 Can_CheckWakeup

[CAN360] ⌈

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.	

⌋()

[CAN361] ⌈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.⌋ ()

[CAN362] ⌈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.⌋()

[CAN363] ⌈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

[CAN233] ⌈

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:	--	

⌋(BSW00312)

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.

[CAN212] ⌈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⌋(BSW01049)

[CAN213] ⌈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:

- The transmission of the L-PDU with higher priority shall not be cancelled and the function Can_Write is left without any actions.
- The function Can_Write shall return CAN_BUSY⌋(BSW01049)

[CAN215] ⌈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:

- 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.
- The function Can_Write shall return CAN_BUSY⌋()

[CAN434] ⌈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:

- 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.
- 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.
- The function Can_Write shall return CAN_BUSY⌋()

[CAN214] ⌈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).⌋(BSW00312, BSW01049)

[CAN275] ⌈The function Can_Write shall be non-blocking.⌋()

[CAN216] ⌈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.⌋()

[CAN217] ⌈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.⌋()

[CAN218] ⌈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.⌋()

[CAN219] ⌈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.⌋()

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.

[CAN443] ⌈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 CAN434_Conf.

[CAN444] ⌈If the L-PDU callout returns false, the L-PDU shall not be processed any further. ⌋()

8.4.2 Enabling/Disabling wakeup notification

[CAN445] ⌈Can driver shall use the following APIs provided by Icu driver, to enable and disable the wakeup event notification:

- `Icu_EnableNotification`
- `Icu_DisableNotification`⌋()

[CAN446] ⌈Icu_EnableNotification shall be called when "external" Can controllers have been transitioned to SLEEP state (CANIF_CS_SLEEP).⌋()

[CAN447] ⌈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.

[CAN431] ⌈If these main functions are called from the BSW Scheduler and the Can module is not initialized, then it shall return immediately without performing any functionality and without raising a production error.⌋(BSW00450)

[CAN110] ⌈There is no requirement regarding the execution order of the CAN main processing functions.⌋(BSW00428)

8.5.1.1 Can_MainFunction_Write

[CAN225] ⌈

Service name:	Can_MainFunction_Write

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

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

⌋()

[CAN031] ⌈The function Can_MainFunction_Write shall perform the polling of TX confirmation and TX cancellation confirmation when CanTxProcessing is set to POLLING.⌋(BSW00432, BSW00373, BSW00376, BSW157)

[CAN178] ⌈The Can module may implement the function Can_MainFunction_Write as empty define in case no polling at all is used.⌋()

[CAN179] ⌈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.⌋()

[CAN441] ⌈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 CAN356_Conf) is supported.」()

8.5.1.2 Can_MainFunction_Read

[CAN226]「

Service name:	Can_MainFunction_Read

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

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

」()

[CAN108]「The function Can_MainFunction_Read shall perform the polling of RX indications when CanRxProcessing is set to POLLING.」(BSW00432, BSW157)

[CAN180]「The Can module may implement the function Can_MainFunction_Read as empty define in case no polling at all is used.」()

[CAN181]「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.」()

[CAN442]「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 CAN358_Conf) is supported.」()

8.5.1.3 Can_MainFunction_BusOff

[CAN227]「

Service name:	Can_MainFunction_BusOff

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

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

」()

[CAN109]「The function Can_MainFunction_BusOff shall perform the polling of bus-off events that are configured statically as 'to be polled'.」(BSW00432, BSW157)

[CAN183]「The Can module may implement the function Can_MainFunction_BusOff as empty define in case no polling at all is used.」()

[CAN184]「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

[CAN228]「

Service name:	Can_MainFunction_Wakeup

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

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

」()

[CAN112]「The function Can_MainFunction_Wakeup shall perform the polling of wake-up events that are configured statically as 'to be polled'.」(BSW00432, BSW157)

[CAN185]「The Can module may implement the function Can_MainFunction_Wakeup as empty define in case no polling at all is used.」()

[CAN186]「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

[CAN368] ⌈

Service name:	Can_MainFunction_Mode
Syntax:	`void Can_MainFunction_Mode( void )`
Service ID[hex]:	0x0c
Timing:	FIXED_CYCLIC
Description:	This function performs the polling of CAN controller mode transitions.

⌋()

[CAN369] ⌈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.⌋()

[CAN379] ⌈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.

[CAN234] ⌈

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).

⌋(BSW00387, BSW01055)

8.6.2 Optional Interfaces

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

[CAN235] ⌈

API function	*Description*
CanIf_CancelTxConfirmation	This service informs CanIf that a L-PDU shall be buffered in CanIf Tx´buffer from CAN hardware object to avoid priority inversion.
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.
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.
Icu_DisableNotification	This function disables the notification of a channel.
Icu_EnableNotification	This function enables the notification on the given channel.

⌋(BSW12056, BSW01054)

8.6.3 Configurable interfaces

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

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

In addition to this section, it is highly recommended to read the documents:
- AUTOSAR Layered Software Architecture [1]
- AUTOSAR ECU Configuration Specification [10]
  This document describes the AUTOSAR configuration methodology and the AUTOSAR configuration metamodel in detail.

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

10.1.1 Configuration and configuration parameters

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

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

10.1.2 Variants

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

10.1.3 Containers

Containers structure the set of configuration parameters. This means:

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

10.1.4 Specification template for configuration parameters

The following tables consist of three sections:

- the general section
- the configuration parameter section
- the section of included/referenced containers

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

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

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

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

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

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

10.2 Containers and configuration parameters

The following chapters summarize all configuration parameters. The detailed meanings of the parameters describe Chapters 7 and Chapter 8.

The described parameters are input for the Can module configurator.

[CAN022] ⌈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.⌋(BSW159)

[CAN047] ⌈The configuration data shall be human readable. ⌋(BSW160)

[CAN024] ⌈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.)⌋(BSW167, BSW12463)

10.2.1 Variants

The Can module provides two variants of configuration sets:

[CAN220] ⌈VARIANT-PRE-COMPILE: Only pre-compile configuration parameters.⌋()

[CAN221] ⌈VARIANT-POST-BUILD: Mix of pre compile- and post build time configuration parameters.⌋()

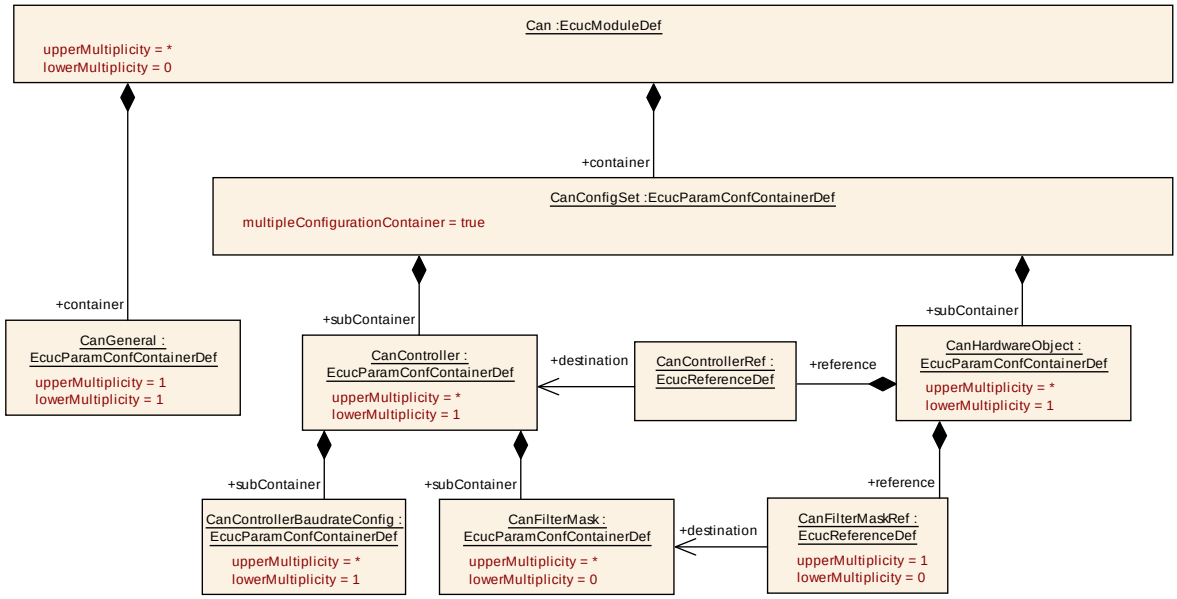


Figure 10-1: Can Module Configuration Layout

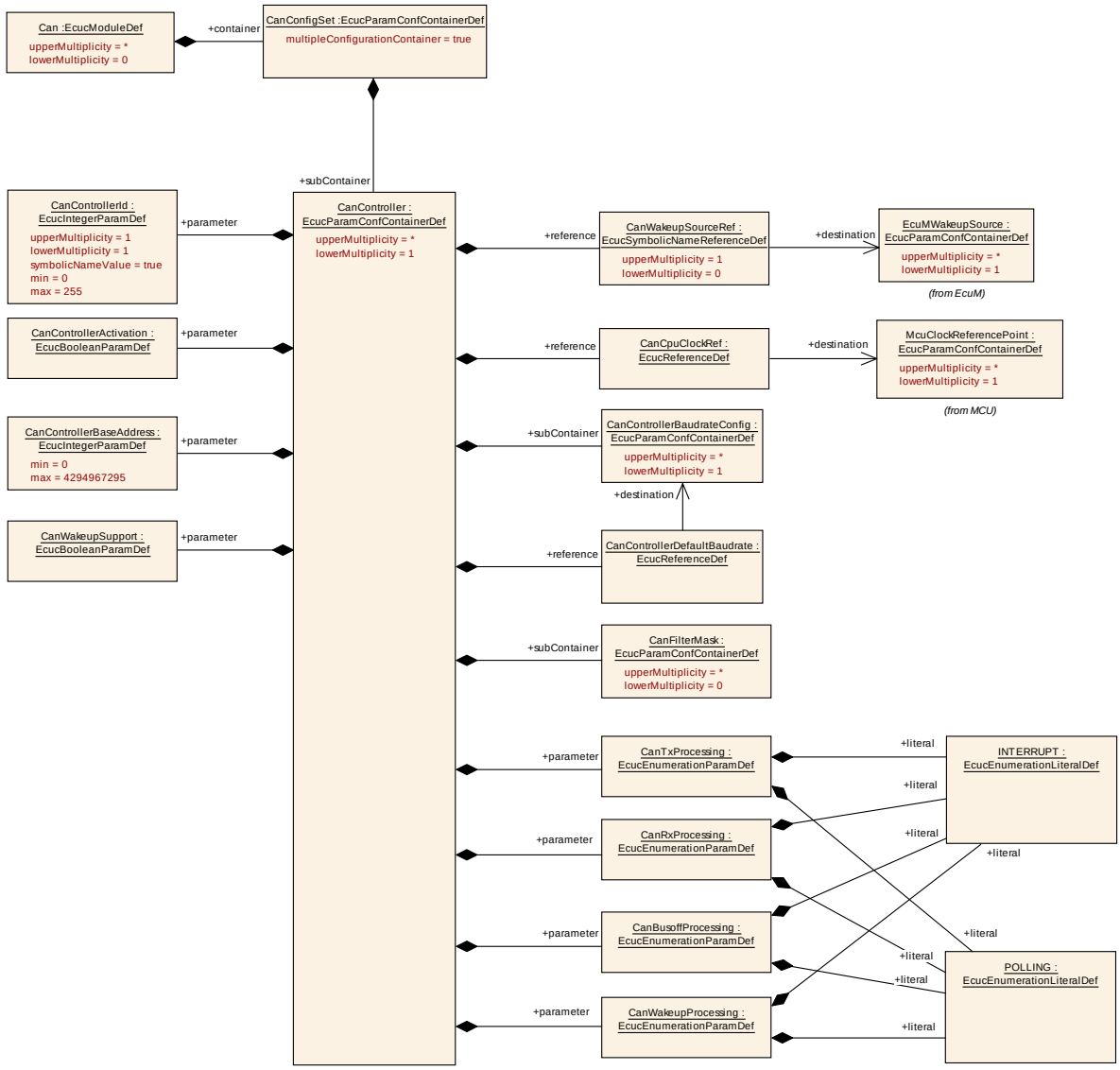


Figure 10-2: Can Controller Configuration Layout

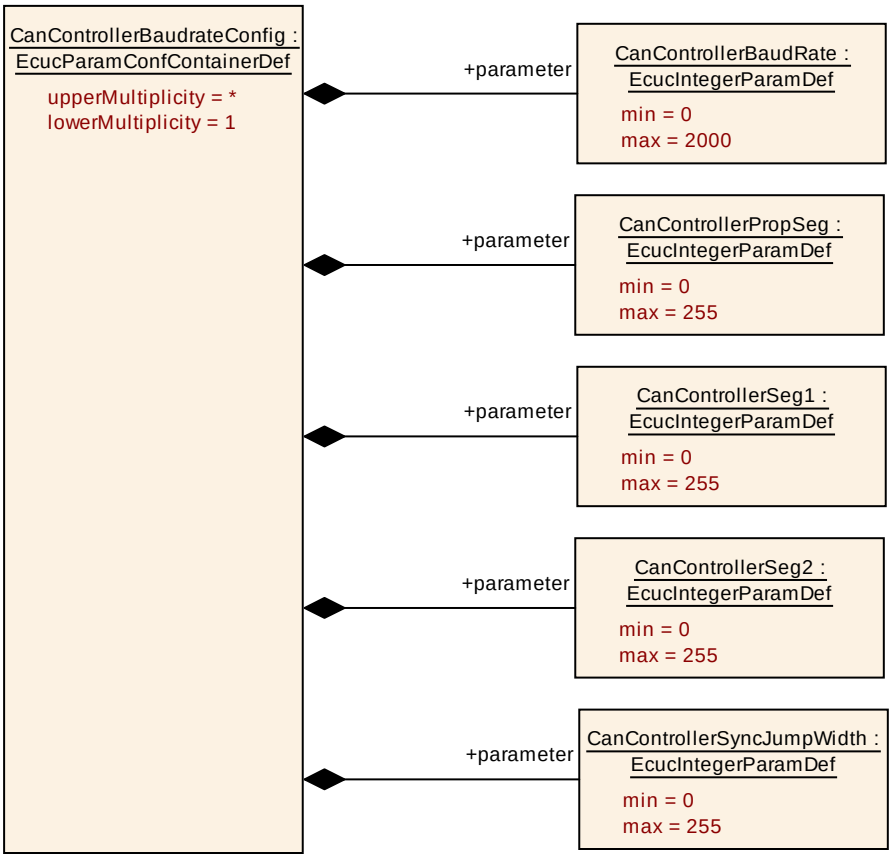


Figure 10-3: Can Controller Baud Rate Configuration Layout

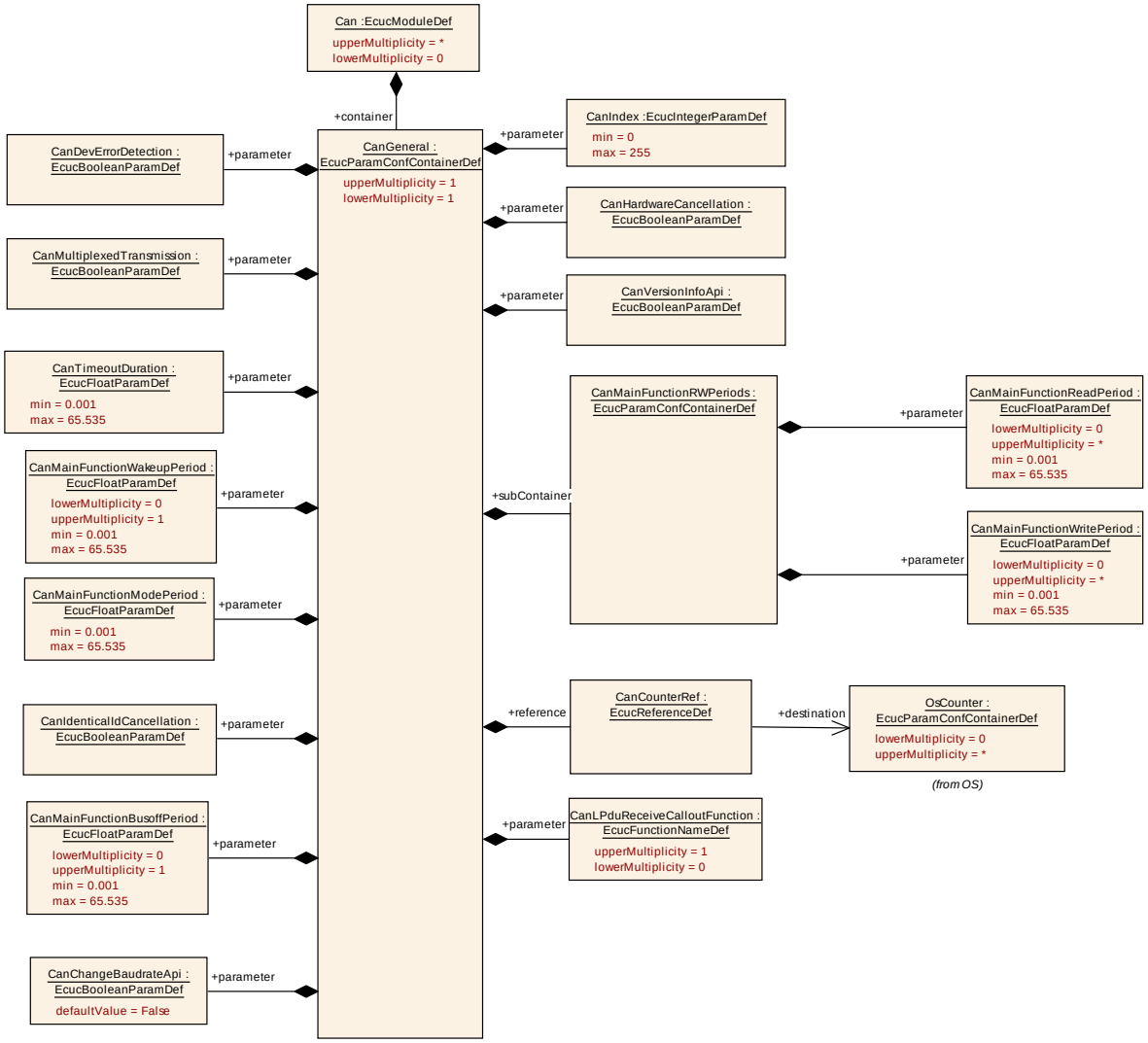


Figure 10-4: Can General Configuration Layout

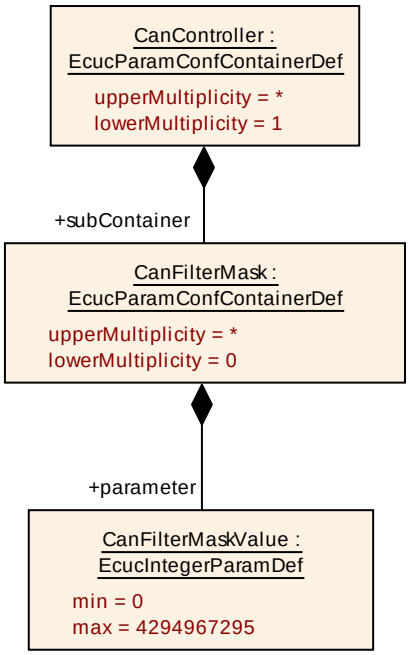


Figure 10-5: Can Filter Mask Configuration Layout

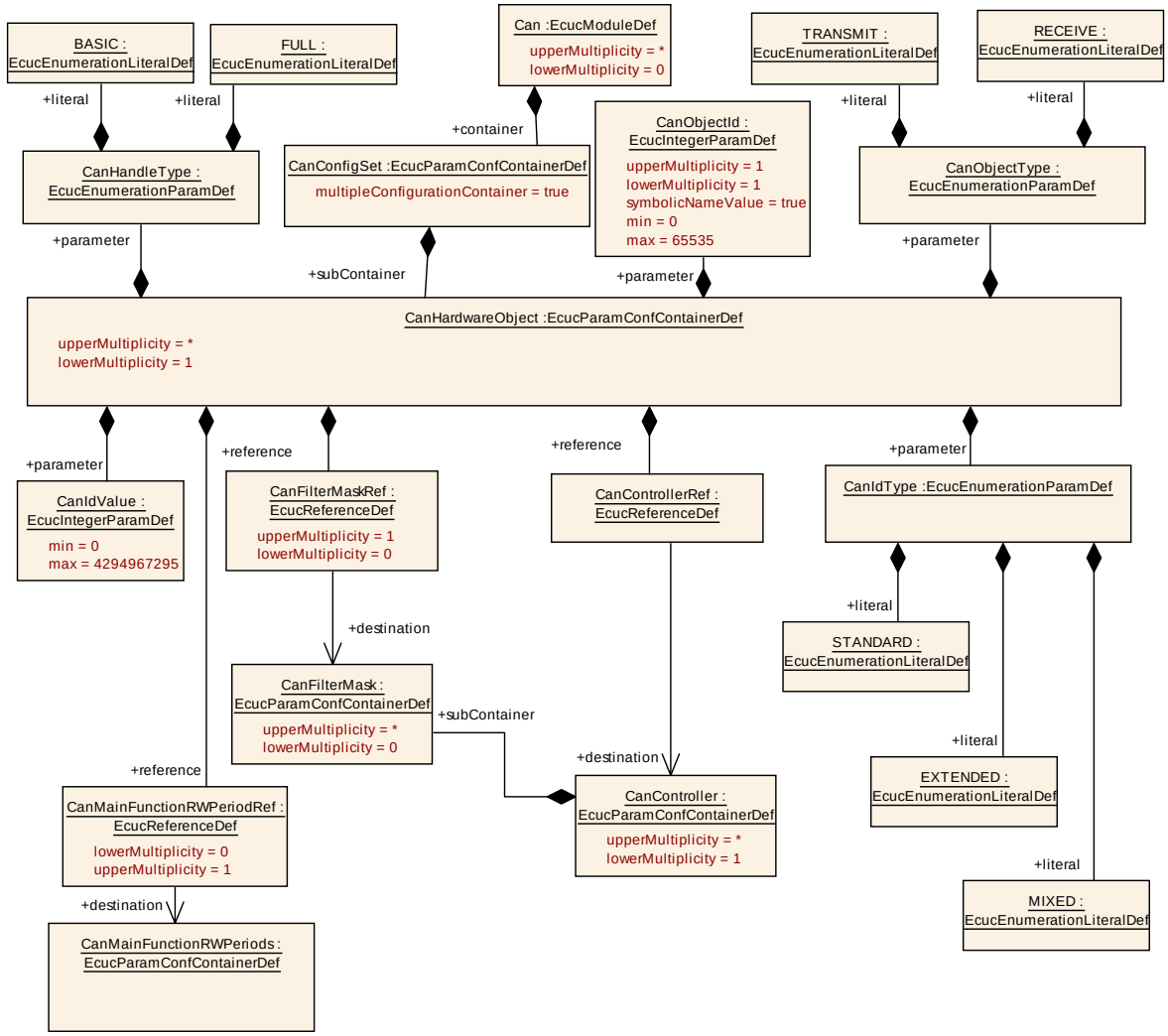


Figure 10-6: Can Hardware Object Configuration Layout

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	**CAN328_Conf :**
Container Name	CanGeneral{CanDriverGeneralConfiguration}
Description	This container contains the parameters related each CAN Driver Unit.
Configuration Parameters	

SWS Item	**CAN436_Conf :**		
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.		
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	**CAN064_Conf :**		
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: Can module		

SWS Item	**CAN069_Conf :**		
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: Can module, CanIf module dependency: CanIf module is configured to support hardware cancellation		

SWS Item	**CAN378_Conf :**		
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: Can module		

SWS Item	**CAN320_Conf :**		
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			

SWS Item	**CAN434_Conf :**		
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: module		

SWS Item	**CAN355_Conf :**		
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	**CAN376_Conf :**		
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	**CAN357_Conf :**		
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	**CAN095_Conf :**		
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: Can module, CanIf module dependency: CAN Hardware Unit supports multiplexed transmission		

SWS Item	**CAN113_Conf :**		
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	0.001 .. 65.535		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Can module		

SWS Item	**CAN106_Conf :**
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: Can module		

SWS Item	**CAN431_Conf :**		
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			

SWS Item	**CAN430_Conf :**		
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			

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanMainFunctionRWPeriods	1	--

10.2.4 CanController

SWS Item	**CAN354_Conf :**
Container Name	CanController{CanController}
Description	This container contains the configuration parameters of the CAN controller(s).
Configuration Parameters	

SWS Item	**CAN314_Conf :**		
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			

SWS Item	**CAN315_Conf :**		
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: Can module		

SWS Item	**CAN382_Conf :**		
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			

SWS Item	**CAN316_Conf :**		
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			

SWS Item	**CAN317_Conf :**		
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			

SWS Item	**CAN318_Conf :**

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			

SWS Item	**CAN319_Conf :**		
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			

SWS Item	**CAN330_Conf :**		
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	**CAN435_Conf :**		
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			

SWS Item	**CAN313_Conf :**		
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			

SWS Item	**CAN359_Conf :**		
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	Reference to [EcuMWakeupSource]		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Can module		

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).
CanTTController	0..1	This container is only included and valid if TTCAN SWS is used and TTCAN is enabled. This container contains the configuration parameters of the TTCAN controller(s) (which are needed in addition to the configuration parameters of the CAN controller(s)). CanTTController is only included, if the controller supports TTCAN.

10.2.5 CanControllerBaudrateConfig

SWS Item	**CAN387_Conf :**
Container Name	CanControllerBaudrateConfig{CanControllerBaudrateConfig}
Description	This container contains bit timing related configuration parameters of the CAN controller(s).
Configuration Parameters	

SWS Item	**CAN005_Conf :**		
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: Can module		

SWS Item	**CAN073_Conf :**
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: Can module		

SWS Item	**CAN074_Conf :**		
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: Can module		

SWS Item	**CAN075_Conf :**		
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: Can module		

SWS Item	**CAN383_Conf :**		
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			

No Included Containers

10.2.6 CanHardwareObject

SWS Item	**CAN324_Conf :**
Container Name	CanHardwareObject{CanHardwareObject}
Description	This container contains the configuration (parameters) of CAN Hardware Objects.

Configuration Parameters

SWS Item	**CAN323_Conf :**		
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: CanIf module 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	**CAN065_Conf :**		
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: Can module, CanIf module		

SWS Item	**CAN325_Conf :**		
Name	CanIdValue {CAN_ID_VALUE}		
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	scope: Can module, CanIf module		

SWS Item	**CAN326_Conf :**
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 are defined under two different name-spaces. 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: Can module, CanIf module		

SWS Item	**CAN327_Conf :**		
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: Can module, CanIf module		

SWS Item	**CAN322_Conf :**		
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			

SWS Item	**CAN321_Conf :**		
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)		
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			

SWS Item	**CAN438_Conf :**		
Name	CanMainFunctionRWPeriodRef {CAN_CONTROLLER_REFERENCE}		
Description	Reference to CAN Controller to which the HOH is associated to.		
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			

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
CanTTHardwareObjectTrigger	0..*	This container is only included and valid if TTCAN SWS is used and TTCAN is enabled. This container contains the configuration (parameters) of TTCAN triggers for Hardware Objects, which are additional to the configuration (parameters) of CAN Hardware Objects. CanTTHardwareObjectTrigger is only included, if the controller supports TTCAN.

10.2.7 CanFilterMask

SWS Item	**CAN351_Conf :**
Container Name	CanFilterMask{CanFilterMask}
Description	This container contains the configuration (parameters) of the CAN Filter Mask(s).
Configuration Parameters	

SWS Item	**CAN066_Conf :**		
Name	CanFilterMaskValue {CAN_FILTER_MASK_VALUE}		
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	scope: Can module, CanIf module dependency: The filter mask settings must be known by the CanIf configuration for optimization of the SW filters.		

No Included Containers

10.2.8 CanConfigSet

SWS Item	**CAN343_Conf :**
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.

10.2.9 CanMainFunctionRWPeriods

SWS Item	CAN437_Conf :
Container Name	CanMainFunctionRWPeriods{CAN_MAIN_FUNCTION_RWPERIODS}
Description	--
Configuration Parameters	

SWS Item	CAN356_Conf :		
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			

SWS Item	CAN358_Conf :		
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			

No Included Containers

10.3 Published Information

[CAN448]「The standardized common published parameters as required by BSW00402 in the SRS General on Basic Software Modules [2] shall be published within the header file of this module and need to be provided in the BSW Module Description. The according module abbreviation can be found in the List of Basic Software Modules [16].」()

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

11 Changes to Release 3

11.1 Deleted SWS Items

SWS Item	*Rationale*
CAN248	Requirement was wrong. CAN174 contains the correct description.
CAN241	Requirement changed into a implementation hint, because it is not a requirement for the Can module.
CAN292, CAN293	Requirement not necessary for these IR functions.
CAN029, CAN081, CAN295, CAN296, CAN297, CAN298, CAN092, CAN093	Requirements removed, because Can module does not report production errors to DEM.
CAN063	Requirement was erroneously re-implemented caused by a wrong metamodel.
CAN247, CAN249	Requirement changed into implementation hint, because these are requirements to the ECU State Manager.
CAN301	Requirement changed into implementation hint, because it is not requirement for the Can module.
CAN054, CAN055	General SPAL requirements BSW12058 and BSW12059 does not exist anymore.
CAN243, CAN421	Requirement changed into implementation hint, because it is not a requirement to the Can module.
CAN176, CAN192, CAN201	Requirement removed, because upper layer modules perform timeout handling.

11.2 Replaced SWS Items

SWS Item	*replaced by SWS Item*	*Rationale*
CAN013	CAN396	CAN013 was lost by document improvement process, is taken back to document and reframed.
CANxxx	CANxxx_Conf	Item name space of all configuration parameters changed to CANxxx_Conf to avoid double defines.
CAN101	CAN402, CAN403	Item split into two items to improve understanding of multiplexed transmission.

11.3 Changed SWS Items

SWS Item	*Rationale*
CAN049	Rewritten to improve understanding.
CAN262, CAN264, CAN266, CAN268	Rewritten to support asynchronous state transition concept.
CAN360, CAN361	Can_Cbk_CheckWakeup renamed to Can_CheckWakeup, because it is not a callback-function. Return type changed from Std_ReturnType to Can_Type.
CAN230	Meaning of return value changed to support asynchronous state transition concept.
CAN330_Conf	Parameter moved from CanGeneral to CanController container.
CAN314_Conf, CAN317_Conf,	Scope and dependency to CanIf module removed. Type changed from derivedEnumerationParamDef to EnumerationParamDef, because there is

CAN318_Conf, CAN319_Conf	no dependency to CanIf module.
CAN257, CAN258	Requirement reframed to improve understanding
CAN355_Conf, CAN356_Conf, CAN357_Conf, CAN358_Conf	Multiplicity changed to 0..1, because parameters are not required when interrupt mode is configured.
CAN321_Conf	Multiplicity changed to 0..1, because CAN Filter mask is not required for transmit objects.
CAN066_Conf	Rewritten to improve understanding
CAN036	Rewritten to be a requirement of Can module.
CAN281	Rewritten to support use of system services
CAN220, CAN221	Rewritten to improve understanding of VARIANT-PRE-COMPILE and VARIANT-POST-BUILD.
CAN286, CAN215	Rewritten to improve understanding of transmit cancellation.
CAN076	Rewritten to improve understanding of multiplexed transmission.
CAN258, CAN267, CAN290	Rewritten to support logical sleep mode.
CAN217, CAN218, CAN219	Missing return value CAN_NOT_OK added in case development error is detected.
CAN060	Rewritten to improve understanding.
CAN286, CAN400, CAN215	Items changed to support configuration of cancellation functionality.

11.4 Added SWS Items

SWS Item	***Rationale***
CAN323_Conf	Id number of CanHandleType changed from CAN324_Conf to CAN323_Conf, because CAN324_Conf was given twice.
CAN365	Item added to support debugging concept.
CAN366	Item added to support debugging concept.
CAN367	Item added to support debugging concept.
CAN368	Item added to support asynchronous state transition concept.
CAN369	Item added to support asynchronous state transition concept.
CAN370	Item added to support asynchronous state transition concept.
CAN371	Item added to support asynchronous state transition concept.
CAN372	Item added to support asynchronous state transition concept.
CAN373	Item added to support asynchronous state transition concept.
CAN376_Conf	Item added to support asynchronous state transition concept.
CAN379	Item added to support asynchronous state transition concept.
CAN398	Item added to support asynchronous state transition concept.
CAN382_Conf	Item added to support mapping between CAN controller ID and CAN controller hardware.
CAN383_Conf	Item added to support CAN controller synchronization jump width.
CAN384	Item added to support asynchronous state transition concept.
CAN385, CAN386	Item added to support naming convention for multiple CAN Driver.
CAN387_Conf	Item added to multiple CAN configuration sets.
CAN388, CAN389, CAN390, CAN391, CAN392, CAN393, CAN394, CAN397	Items added to describe header file structure.
CAN395	Item added to support CAN_E_DATALOST development error detection.
CAN399, CAN400	Item added to support transmit cancellation.
CAN401	Item added to support multiplexed transmission.
CAN404, CAN405	Item added to support logical sleep mode.
CAN236, CAN237	Item added to limit the support of remote frames.
CAN016	Item added to support [BSW01051] Transmit Confirmation

CAN406	Item added to support [BSW00435] Module Header File Structure for the Basic Software Scheduler
CAN407	Item added to support [BSW12461] Responsibility for register initialization
CAN408, CAN409, CAN410, CAN411, CAN412	Item added to improve description of invalid state transitions.
CAN413, CAN414, CAN415, CAN416, CAN417	Formal item IDs given to the types defined in section 8.2.
CAN418	CAN037 split into atomic SWS items: CAN037 and CAN418
CAN419, CAN420	CAN033 split into atomic SWS items: CAN033, CAN419 and CAN420
CAN422	CAN260 split into atomic SWS items: CAN260 and CAN422
CAN423	CAN279 split into atomic SWS items: CAN279 and CAN423
CAN424	CAN027 split into atomic SWS items: CAN027 and CAN424
CAN425	CAN196 split into atomic SWS items: CAN196 and CAN425
CAN426	CAN197 split into atomic SWS items: CAN197 and CAN426
CAN427, CAN428	Plain text converted into SWS item.
CAN429	Item added for new Can_HwHandle_Type
CAN430_Conf	Item added to support TTCAN.
CAN431	Item added to support [BSW00450] Main Function Processing for Un-Initialized Modules.
CAN378_Conf, CAN432, CAN433, CAN434	Items added to support configuration of cancellation functionality.
CAN001_PI	Rework of Published Information

12 Not applicable requirements

[CAN999]「These requirements are not applicable to this specification.」(BSW170, BSW00383, BSW00395, BSW00397, BSW00398, BSW00399, BSW00400, BSW168, BSW00423, BSW00424, BSW00425, BSW00426, BSW00427, BSW00429, BSW00433, BSW00336, BSW00422, BSW00417, BSW00409, BSW00455, BSW162, BSW00415, BSW00325, BSW00326, BSW00342, BSW00453, BSW00413, BSW00307, BSW00447, BSW00353, BSW00361, BSW00439, BSW00449, BSW00378, BSW00359, BSW00440, BSW00443, BSW00444, BSW00445, BSW00446, BSW12163, BSW12462, BSW12068, BSW12064, BSW01125, BSW01126)