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

Document Version	2.4.0
Document Status	Final
Part of Release	3.1
Revision	5

Document Change History

Date	Version	Changed by	Change Description
20.09.2010	2.4.0	AUTOSAR Administration	• Updated CAN271 and CAN234 • Legal disclaimer revised
28.01.2010	2.3.0	AUTOSAR Administration	• Description of Multiplexed Transmit Functionality improved. • Reference to CanIf_SetWakeupEvent replaced by EcuM_CheckWakeup. • Added missing literal specification for CanBusoffProcessing, CanRxProcessing, CanTxProcessing, CanWakeupProcessing • SchM_Can.h included in File Structure • Create new CAN artefacts with updated BSW UML Model • 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

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.

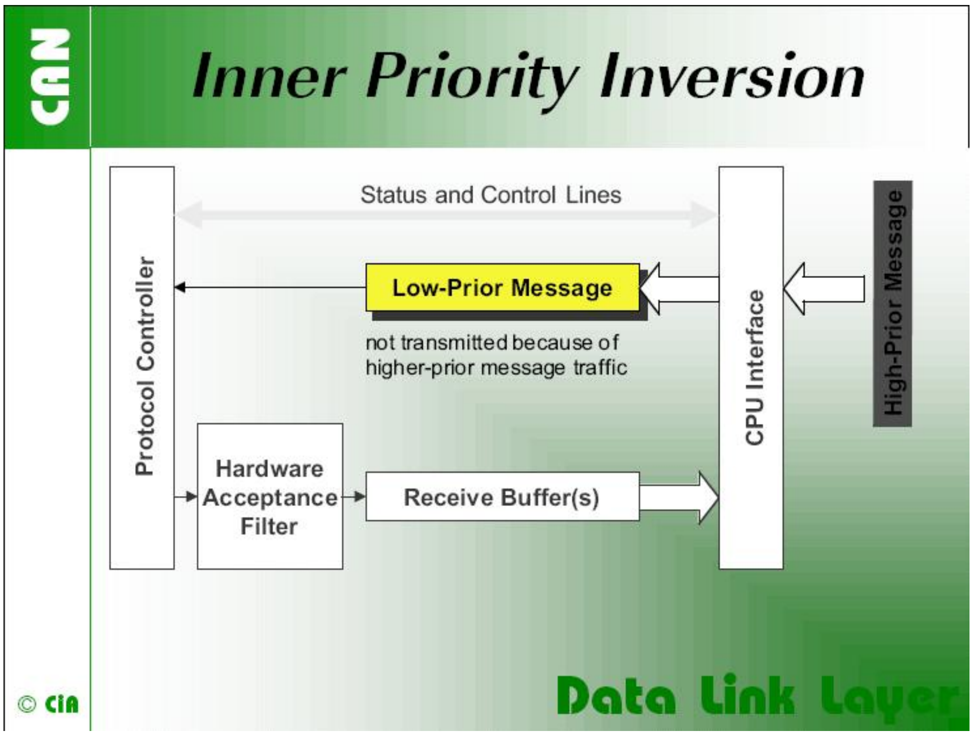
Several CAN controllers can be controlled by a single Can module as long as they belong to the same CAN Hardware Unit.

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

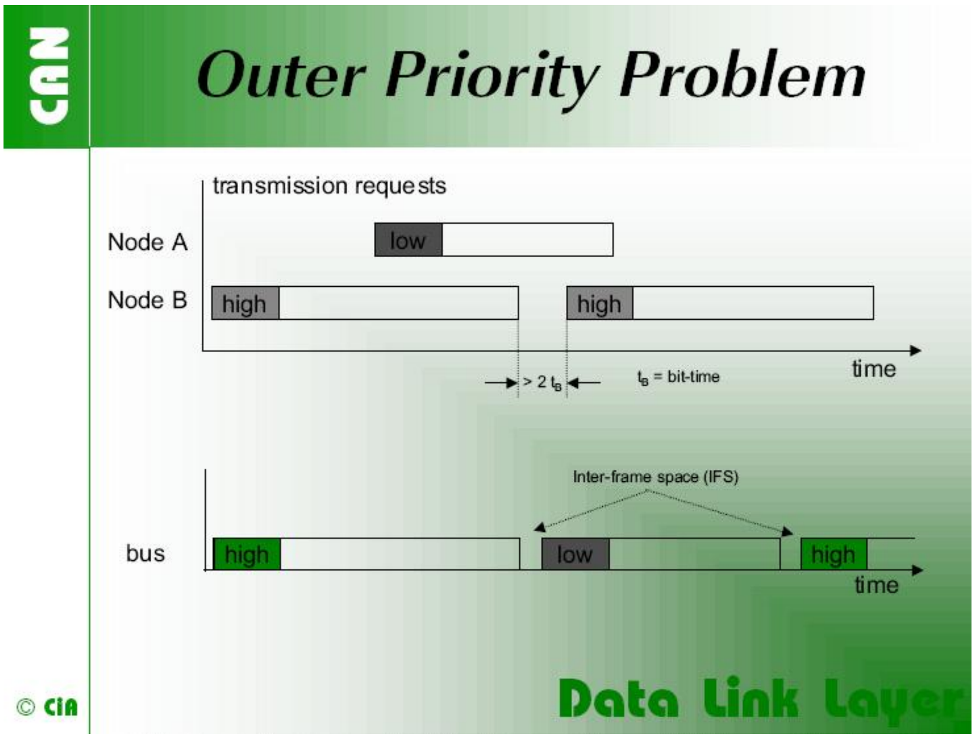
Abbreviation / Acronym:	***Description:***
CAN controller	A CAN controller serves exactly one physical channel.
CAN Hardware Unit	A CAN Hardware unit may consist of one or multiple CAN controllers of the same type and one, two or multiple CAN RAM areas. The CAN hardware unit is located on-chip or as external device. The CAN hardware unit is represented by one CAN driver. 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 [15])
CAN L-SDU	Data Link Layer Service Data Unit. Data that is transported inside the L-PDU. (see [15])
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. Typically each HRH represents exactly 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. Typically each HTH represents one or several (only Release 2) hardware objects, that are configured as hardware transmit 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)



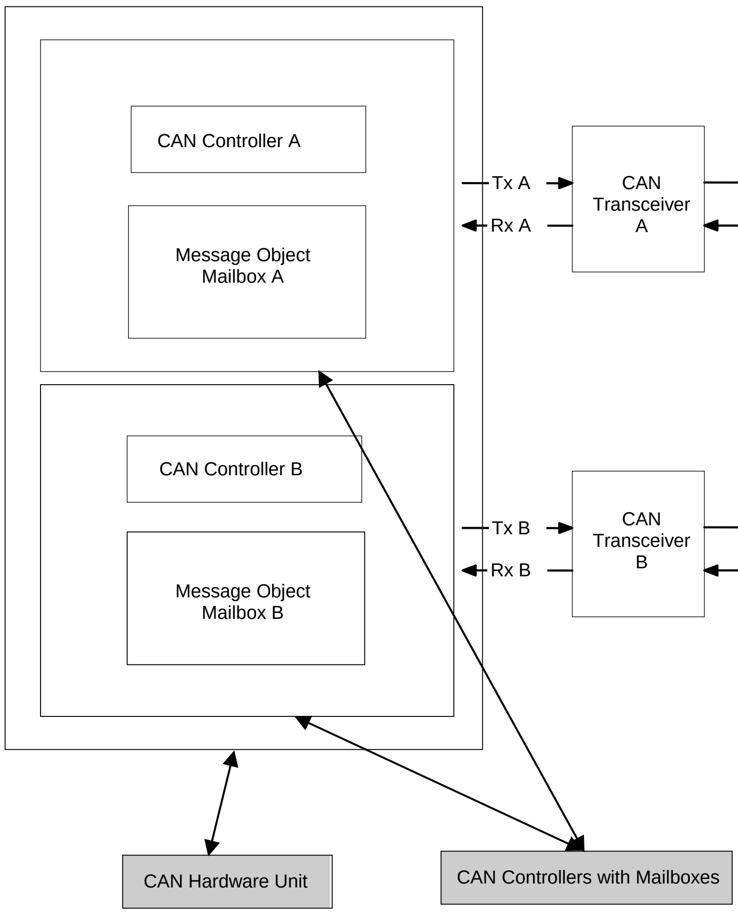
"The problem of outer priority inversion may occur in some CAN implementations. Let us assume that a CAN node wishes to transmit a package of consecutive messages with high priority, which are stored in different message buffers. If the interframe space between these messages on the CAN network is longer 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
Tx A
Rx A
CAN Transceiver A
CAN Bus A
Physical Channel A
CAN Controller B
Message Object Mailbox B
Tx B
Rx B
CAN Transceiver B
CAN Bus B
Physical Channel B
CAN Hardware Unit
CAN Controllers with Mailboxes

3 Related documentation

3.1 Input documents

[1] Layered Software Architecture
AUTOSAR_LayeredSoftwareArchitecture.pdf

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

[3] General Requirements on SPAL
AUTOSAR_SRS_SPAL_General.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_DET.pdf

[7] Specification of ECU State Manager
AUTOSAR_SWS_ECU_StateManager.pdf

[8] Specification of MCU Driver
AUTOSAR_SWS_MCU_Driver.pdf

[9] Specification of Operating System
AUTOSAR_SWS_OS.pdf

[10] Specification of ECU Configuration
AUTOSAR_ECU_Configuration.pdf

[11] Specification of C Implementation Rules
AUTOSAR_SWS_C_ImplementationRules.pdf

[12] Specification of ECU State Manager
AUTOSAR_SWS_ECU_StateManager.pdf

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

3.2 Related standards and norms

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

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

[16] 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.
The only exception is when the hardware supports the 'merging' of several controllers to one. Then these 'merged' controllers are represented as one controller by the Can module.

CAN237: The Can module does not support CAN Remote Frames. The Can module shall not process received remote frames.

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: f the CAN controller is on-chip, the Can module shall not use any service of other drivers.

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.

CAN240: The Mcu module (SPAL see [8]) shall configure register settings that are 'shared' with other modules

CAN241: The Can module's environment shall make sure that the Mcu module is inititalized 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 (i.e. SPI).

CAN243: If the Can module uses services of other MCAL drivers (e.g. SPI), the Can module's environment shall make sure 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.

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 contain a timeout detection in case the hardware doesn't react in the expected time (hardware error) to prevent endless loops. As long

[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.

as the system service does not provide a free running timer this timeout shall be realized with a fixed number of loops.[4]

Reason: 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, so the CAN main functions can't be used for that purpose.

In case consistency concepts (resources/critical sections) are offered by the BSW Module Scheduler, the according services will be used by the Can module.

5.1.4 Can module Users

CAN058: The Can module interacts among other modules (eg. Diagnostic Event Manager (DEM), Development Error Tracer (DET)) 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.

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.

5.2.2 Header file structure

CAN034:

[4]In future specifications the System Services will provide two services with ticks of different resolutions. These ticks will be used to prevent endless loops due to hardware malfunction.

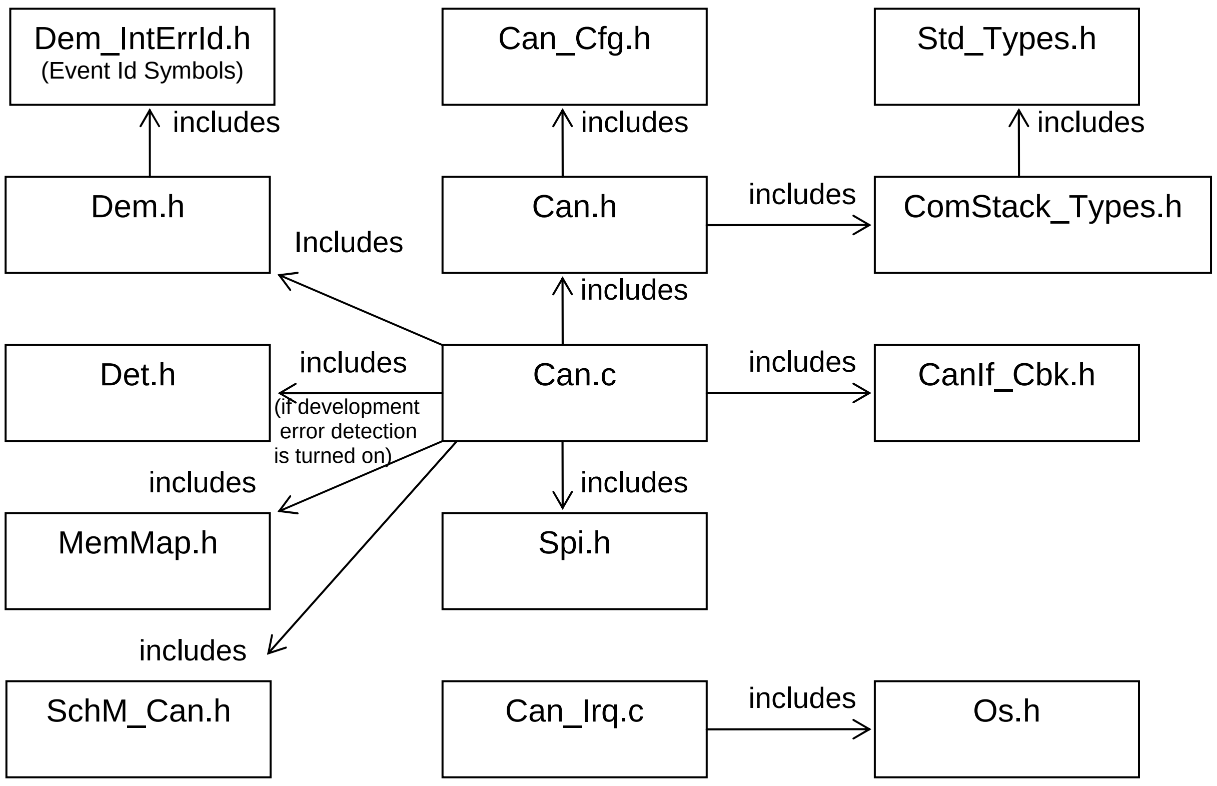


Figure 5-1: File structure for the Can module

CAN035: The module Can_Irq.c contains the implementation of interrupt frames [BSW00314]. The implementation of the interrupt service routine shall be in Can.c

CAN036: The header file CanIf_Cbk.h contains the declarations of the callback functions imported by the modules calling the callbacks.
The Can module does not provide callback functions (no Can_Cbk.h, see also CAN244)

CAN043: The file Can.h contains the declaration of the Can module API

CAN037: The file Can.h only contains 'extern' declarations of constants, global data, type definitions and services that are specified in the Can module SWS.
Constants, global data types and functions that are only used by the Can module internally, are declared in Can.c

CAN404: The Can module shall include the header file SchM_Can.h in order to access the module specific functionality provided by the BSW Scheduler.

6 Requirements traceability

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	chapter 10 The configuration parameters are described in a general way. they can be simply transformed into #defines. Generated code will not contain those defines. The code generator will process e.g. a XML file“
[BSW159] Tool-based configuration	CAN022
[BSW167] Static configuration checking	CAN023, CAN024
[BSW171] Configurability of optional functionality	CAN064, CAN095, CAN069
[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	fulfilled by parameter definitions in Chapter 10.2
[BSW00391] Parameter shall have unique names	fulfilled by parameter definitions in Chapter 10.2
[BSW00392] Parameters shall have a type	fulfilled by parameter definitions in Chapter 10.2
[BSW00393] Parameters shall have a range	fulfilled by parameter definitions in Chapter 10.2
[BSW00394] Specify the scope of the parameters	fulfilled by parameter definitions in 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	fulfilled by parameter definitions in 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.
[BSW00402] Published information	CAN085
[BSW00375] Notification of wake-up reason	CAN018

[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
[BSW00407] Function to read out published parameters	CAN105, CAN106
[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)
[BSW00431] The BSW Scheduler module implements task bodies	not applicable (requirement on the BSW scheduler module)
[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)
[BSW00434] The Schedule Module shall provide an API for exclusive areas	not applicable (requirement on schedule module)
[BSW00336] Shutdown interface	not applicable
[BSW00337] Classification of errors	CAN026, CAN027, CAN028, CAN029
[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	CAN029, CAN113
[BSW00421] Reporting of production relevant error events	CAN029
[BSW00422] Debouncing of production relevant error status	not applicable (requirement on the DEM)
[BSW00420] Production relevant error event rate detection	not applicable (requirement on the DEM)
[BSW00417] Reporting of Error Events by Non-Basic Software	not applicable (this is a BSW mdoule)
[BSW00323] API parameter checking	CAN026
[BSW004] Version check	CAN111
[BSW00409] Header files for production code error IDs	CAN081

[BSW00385] List possible error notifications	CAN104
[BSW00386] Configuration for detecting an error	CAN089
[BSW161] Microcontroller abstraction	see Chapter 1
[BSW162] ECU layout abstraction	not applicable (done in CanIf module)
[BSW00324] Do not use HIS Library	Fulfilled by the concept of Can module and 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)
[BSW166] BSW Module interfaces	CAN043
[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	CAN063
[BSW160] Human-readable configuration data	CAN047
[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 (his requirement is fulfilled by the CanIf module specification)
[BSW00347] Naming separation of drivers	CAN077
[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 7.8 error names have been selected accordingly
[BSW00335] Status values naming convention	chapter 7.1 is fulfilled by state description
[BSW00350] Development error detection keyword	CAN064
[BSW00408] Configuration parameter naming convention	fulfilled by parameter definitions in Chapter 10.2
[BSW00410] Compiler switches shall have defined values	fulfilled by parameter definitions in Chapter 10.2
[BSW00411] Get version info keyword	CAN106
[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, CAN404

[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	see Chapter 1
[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
[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
[BSW00436] Module Header File Structure for the Basic Software Memory Mapping	CAN034
[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	CAN234
[BSW12267] Configuration of wake-up sources	CAN257, CAN258, CAN018
[BSW12057] Driver module initialization	CAN154
[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)
[BSW12058]] Individual initialization of overall registers	CAN054
[BSW12059] General initialization of overall registers	CAN055
[BSW12060] Responsibility for initialization of one-time writable registers	CAN055
[BSW12062] Selection of static configuration sets	CAN056
[BSW12068] MCAL initialization sequence	not applicable (requirement on station manager)
[BSW12069] Wake-up notification of ECU State Manager	CAN018
[BSW157] Notification mechanisms of drivers and handlers	CAN026, CAN028, CAN029, CAN031, CAN108, CAN109, CAN112
[BSW12155] Prototypes of callback functions	not applicable (information has to be exchanged (see [BSW00359], [BSW00360]))
[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, CAN258, CAN018
[BSW12077] Non-blocking implementation	CAN029
[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	done in chapter 10
[BSW12081] Use HIS requirements as input	No requirement This req. does not affect the HIS Can module

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	see Chapter 1
[BSW01035] Multiple CAN controller support	see Chapter 1
[BSW01036] CAN Identifier Length Configuration	CAN065
[BSW01037] Hardware Filter Configuration	CAN066, CAN325
[BSW01038] Bit Timing Configuration	CAN005, CAN063, CAN073, CAN074, CAN075
[BSW01039] CAN Hardware Object Handle definitions	CAN324

[BSW01040] HW Transmit Cancellation configuration	CAN069
[BSW01058] Configuration of multiplexed transmission	CAN095
[BSW01062] Configuration of polling mode	CAN007
[BSW01135] Configuration of multiple TX Hardware Objects	CAN100
[BSW01041] Can module Module Initialization	CAN154
[BSW01042] Selection of static configuration sets	CAN062
[BSW01043] Enable/disable Interrupts	CAN049, CAN050
[BSW01059] Data Consistency	CAN011, CAN012
[BSW01045] Reception Indication Service	CAN013
[BSW01049] Dynamic transmission request service	CAN212, CAN213, CAN214
[BSW01051] Transmit Confirmation	CAN016
[BSW01053] CAN controller mode select	CAN017
[BSW01054] Wake-up Notification	CAN018
[BSW01132] Mixed mode for notification detection on CAN HW	CAN099
[BSW01133] HW Transmit Cancellation Support	CAN285, CAN286, CAN287, CAN288, CAN399, CAN400
[BSW01134] Multiplexed Transmission	CAN277, CAN401, CAN402, CAN403, CAN076
[BSW01055] Bus-off Notification	CAN019
[BSW01060] no automatic bus-off recovery	CAN020
[BSW01122] Support for wakeup during sleep transition	CAN048

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 CanIf module interface 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. The Can module shall disable all unused interrupts in the CAN controller. The Can module shall reset the interrupt flag at the end of the ISR (if not done automatically by hardware). 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].

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.

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 API name>_<vendorID>_<driver abbreviation>()
```

BSW00347 specifies the naming convention.
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, which is shown in Figure 7.1.

CAN103: After power-up/reset, the Can module shall be in the state CAN_UNINIT.

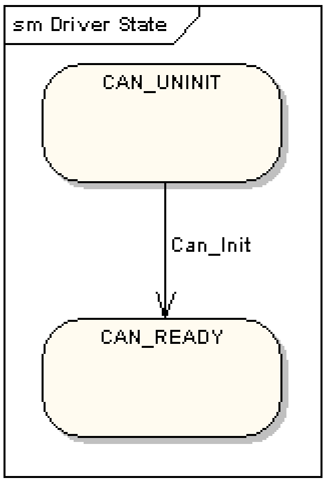


Figure 7-1

CAN245: The function Can_Init shall initialize all CAN controllers according to their configuration.

Each CAN controller must then be started separately by calling the function Can_SetControllerMode(CAN_T_START).

CAN246: After initializing all controllers inside the HW Unit, the function Can_Init shall change the module state to CAN_READY.

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.

CAN247: The Can module's environment shall call Can_Init at most once during runtime.

CAN248: The function Can_Init shall report the error CAN_E_UNINIT, if Can_Init was called prior to any Can module function.

Implementation hint:
The Can module must only implement a variable for the module state, when the development error tracing is switched on. When the development error tracing is switched off, the Can module does not need to implement this 'state machine', because the state information is only needed to check if Can_Init was called prior to any Can module function.

7.3 CAN Controller State Machine

Each CAN controller has a state machine implemented in hardware.
For each CAN controller a 'software' state machine is implemented in the CanIf module. [5] shows the implemented 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_InitController 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 these state changes by external events:

- Bus-off
- HW wakeup

These 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 the CanIf module to trigger only transitions that are allowed in the current state. Only when development errors are enabled, does the Can module check the transition and, in case of wrong implementation of the CanIf module, raise 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 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.

CANIF_CS_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.

CANIF_CS_STOPPED

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

CANIF_CS_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.

CANIF_CS_SLEEP

The hardware settings only differ from CANIF_CS_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, when transitioning into mode “CANIF_CS_SLEEP”, the Can module shall set the controller to a state from which the hardware can be woken over CAN Bus.

CAN258: When the CAN hardware does not support sleep mode, the Can module shall use the same hardware state for CANIF_CS_SLEEP as for CANIF_CS_STOPPED.

7.3.2 State Transitions

A state transition is triggered by software with the function Can_SetControllerMode, with the required transition as parameter. Except for CAN_T_SLEEP, this function is non-blocking.
Some transitions are triggered by events on the bus (hardware). These transitions cause a notification by means of a callback function.
Typically, for state transitions the CAN controller configuration is changed.
Plausibility checks for state transitions are only performed with development error detection switched on. The behavior for invalid[5] transitions in production code is undefined.

Can_Init

- CANIF_CS_UNINIT -> CANIF_CS_STOPPED (for all controllers in HW unit)

[5] Example for invalid transition: CAN_T_SLEEP when controller state is CAN_CS_STARTED

- 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 CANIF_CS_STOPPED.

Can_InitController

- CANIF_CS_STOPPED -> CANIF_CS_STOPPED
- software triggered by the function call Can_InitController
- changes the CAN controller configuration

All control registers are set according to the static configurations that are not global CAN HW Unit settings (See also Can_Init).

CAN256: The Can module's environment shall only call Can_InitController when the CAN controller is in state CANIF_CS_STOPPED.

CAN260: The function Can_InitController shall maintain the CAN controller in the state CANIF_CS_STOPPED. The function Can_InitController shall ensure that any settings that will cause the CAN controller to participate in the network are not set.

Can_SetControllerMode(CAN_T_START)

- CANIF_CS_STOPPED -> CANIF_CS_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 be non-blocking and shall not wait until the CAN controller is fully operational.

Transmit requests that are initiated before the CAN controller is operational may either be delayed or 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.

Can_SetControllerMode(CAN_T_STOP)

- CANIF_CS_STARTED -> CANIF_CS_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 be non-blocking and shall not wait until the CAN controller is really switched off.

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.

Can_SetControllerMode(CAN_T_SLEEP)

- CANIF_CS_STOPPED -> CANIF_CS_SLEEP
- software triggered

CAN265: The function Can_SetControllerMode(CAN_T_SLEEP) shall put the controller into sleep mode.

CAN266: If the CAN HW does support a sleep mode, the function Can_SetControllerMode(CAN_T_SLEEP) shall be blocking and shall only return when it is assured that the CAN hardware is wakeable.

CAN290: If the CAN HW does not support a sleep mode, the function Can_SetControllerMode(CAN_T_SLEEP) shall have no effect (as the controller is already in stopped state).

Can_SetControllerMode(CAN_T_WAKEUP)

- CANIF_CS_SLEEP -> CANIF_CS_STOPPED
- software triggered

CAN267: If the CAN HW does not support a sleep mode, the function Can_SetControllerMode(CAN_T_WAKEUP) shall have no effect (as the controller is already in stopped state).

CAN268: The function Can_SetControllerMode(CAN_T_WAKEUP) shall be non-blocking.

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

- CANIF_CS_SLEEP -> CANIF_CS_STOPPED
- triggered by incoming L-PDUs

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 module shall transition into the state CAN_IF_CS_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.

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.

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

CAN020:
- CANIF_CS_STARTED -> CANIF_CS_STOPPED
- triggered by hardware if the CAN controller reaches bus-off state
- The CanIf module is notified with the callback function CanIf_ControllerBusOff after stopped state is reached.

CAN272: After bus-off detection, the Can module shall transition to the state CANIF_CS_STOPPED and shall ensure that the CAN controller doesn't participate on the network anymore.

CAN273: After bus-off detection, the Can module shall cancel still pending messages without raising a cancellation notification.

CAN274: The Can module shall disable or suppress automatic bus-off recovery

7.4 Can module/Controller Initialization

CAN249: The CanIf 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

CAN053: registers of CAN controller Hardware resources that are not used.

CAN054: registers that contain ‘overall’ settings also relevant for other driver modules (i.e. SPAL) in a way that other modules are not affected (BSW12058). Can_Init shall perform write access to these registers in an atomic manner.

CAN055: registers that contain ‘overall’ settings also relevant for other driver modules that cannot be separated from each other (these are initialized by a system module of the microcontroller abstraction layer) (BSW12059).

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).

CAN062: The function Can_InitController shall re-initialize the CAN controller and the controller specific settings.

The CanIf module must first set the CAN controller in CANIF_CS_STOPPED state. Then it may call Can_InitController.

CAN255: The function Can_InitController 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.

CAN291: Config is a pointer into an array of hardware specific data structure stored in ROM.The different controller configuration sets are located as data structures in ROM.

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.

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 7-2 for a possible configuration.

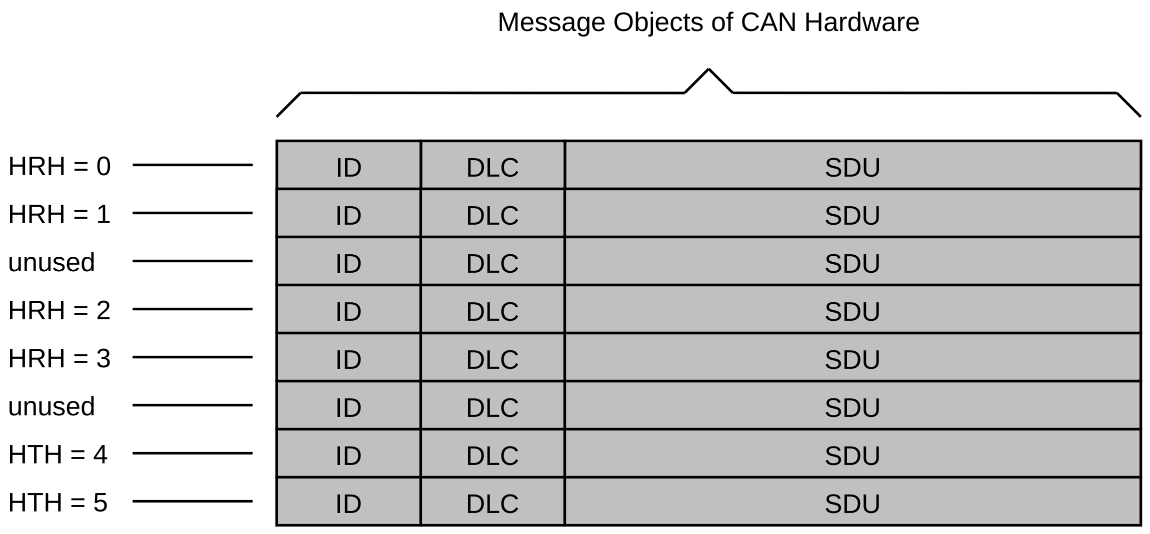


Figure 7-2: Example of assignment of HTHs and HRHs to the Hardware Objects. The numbering of HTHs and HRHs are implementation specific. The chosen numbering is only an example.

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.

7.5.1 Priority Inversion

To prevent priority inversion two mechanisms are necessary: multiplexed transmit 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.

CAN401: Several transmit hardware objects shall be assigned by one HTH to represent one transmit entity to the upper layer.

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.

CAN403: The Can module shall support multiplexed transmission for devices, which send L-PDUs in order of L-PDU priority.

CAN076: The Can module shall NOT support software emulation for the transmission in order of LPDU-priority.

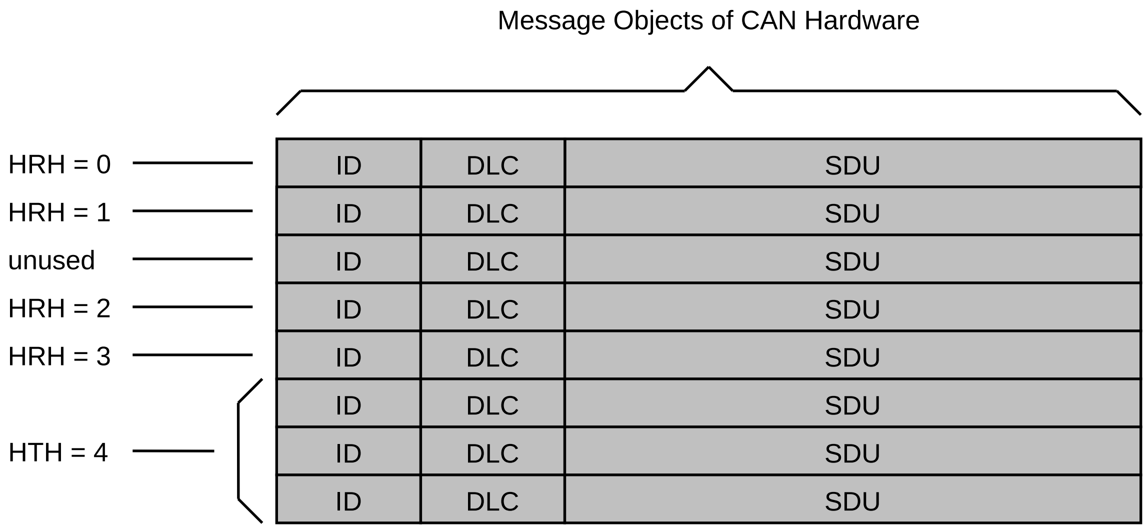


Figure 7-3: Example of assignment of HTHs and HRHs to the Hardware Objects with multiplexed transmission. The numbering of HTHs and HRHs are implementation specific. The chosen numbering is only an example.

7.5.1.2 Transmit Cancellation

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

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

CAN285: Transmit cancellation may only be used when transmit buffers are enabled inside the CanIf module.

CAN286: The Can module shall initiate a cancellation, when the hardware transmit object assigned by a HTH is busy and an L-PDU with the identical or higher priority is requested to be transmitted.

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.

CAN400: The Can module shall initiate a cancellation, when one of the hardware transmit objects assigned by the HTH is busy and an L-PDU with identical priority is requested to be transmitted.

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.

CAN288: The TX request for the new L-PDU shall be repeated by the CanIf module, inside the notification function CanIf_CancelTxConfirmation.

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).

7.6 L-PDU reception

CAN279: On L-PDU reception, the Can module shall call the RX indication callback function with ID, DLC and pointer to the L-SDU buffer as parameter. If necessary, the Can module shall convert the ID and DLC to a standardized format (i.e. MSB that marks extended identifiers).

CAN060: 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.
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.

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: heguarantee 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.

If the hardware can't 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.

CAN301: The configuration check shall assure that the interrupt latency or Can_MainFunction_Read call period can't exceed the time for the reception of one L-PDU.

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.

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_Cbk_CheckWakeup.

When wakeup events are detected by polling, the ECU State Manager will cyclically call Can_Cbk_CheckWakeup via the Can Interface as before. In both cases, Can_Cbk_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 [12].

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.

CAN007: It shall be possible to configure the driver such that no interrupts at all are used (complete polling).

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. This must be ensured by the calling CanIf module. 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)

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
Timeout caused by hardware error	Production	CAN_E_TIMEOUT	Assigned by DEM

7.10.1 Development Errors

CAN026: shall indicate errors that are caused by erroneous usage of the Can module API. This covers API parameter checks and call sequence errors.

CAN028: call the Development Error Tracer when DET is switched on and the Can module detects an error.

CAN091: After return of the DET the Can module's function that raised the development error shall return immediately.

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.

CAN080: Development error values are of type uint8.

7.10.2 Production Errors

CAN029: call the central error function of the Diagnostic Event Manager if the Can module detects hardware errors or failures.
The Syntax for the function call is Dem_ReportErrorStatus(EventId, EventStatus).
The only error that is reported to DEM by the Can module is CAN_E_TIMEOUT.

Depending on the CAN hardware, a change of setting may take over only after a delay.

CAN295: In that case, the Can module shall poll a flag of the CAN status register until the flag signals that the change takes affect and then return.

CAN296: This polling shall take only a (configurable) limited time and thus number of poll cycles is limited.

CAN297: When this time is elapsed the Can module shall raise the error code CAN_E_TIMOUT.

CAN298: In case of a CAN_E_TIMEOUT error the COM Stack must be re-initialized or the COM functionality must be switched off.

CAN081: Values for production code Event Ids are assigned externally by the configuration of the Dem. They are published in the file Dem_IntErrId.h and included via Dem.h.

CAN092: After return of DEM the function of the Can module that raised the production error shall return immediately.

CAN093: The function of the Can module which provides a return value and which raised a production error shall return with CAN_NOT_OK.

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). No code for catching development errors shall be generated, when development errors are switched off.

7.13 Version Check

CAN111: Can.c shall check if the correct version of Can.h is included. This shall be done by a preprocessor check of the version numbers CAN_SW_MAJOR_VERSION, CAN_SW_MINOR_VERSION and CAN_SW_PATCH_VERSION.

8 API specification

The prefix of the function names may be changed in an implementation with several Can modules as described in **CANIF124** in [5].

8.1 Imported types

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

CAN222:

Header file	***Imported Type***
Dem_Types.h	Dem_EventIdType
CanIf_Types.h	CanIf_WakeupSourceType
Std_Types.h	Std_VersionInfoType
	Std_ReturnType
ComStack_Types.h	PduIdType

8.2 Type definitions

8.2.1 Can_ConfigType

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_ControllerConfigType

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

Can_PduType

Name:	`Can_PduType`		
Type:	`Structure`		
Element:	`uint8*`	`sdu`	--
	`Can_IdType`	`id`	--
	`PduIdType`	`swPduHandle`	

	`uint8`	`length`	--
Description:	This type is used to provide ID, DLC and SDU from CAN interface to CAN driver.		

8.2.3 Can_IdType

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

8.2.4 Can_StateTransitionType

Name:	`Can_StateTransitionType`	
Type:	`Enumeration`	
Range:	`CAN_T_START`	--
	`CAN_T_STOP`	--
	`CAN_T_SLEEP`	--
	`CAN_T_WAKEUP`	--
Description:	State transitions that are used by the function CAN_SetControllerMode	

8.2.5 Can_ReturnType

CAN039:

Name:	`Can_ReturnType`	
Type:	`Enumeration`	
Range:	`CAN_OK`	success
	`CAN_NOT_OK`	error occured 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 .	

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.	

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

CAN176: The function Can_Init shall raise the error CAN_E_TIMEOUT if the initialization could not be performed (indicates defective hardware).

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 'uninitialized' state.

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:	Non 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).

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_GetGetVersionInfo shall be pre compile time configurable `On/Off` by the configuration parameter: `CanVersionInfoApi.`

8.3.2 Services affecting one single CAN Controller

8.3.2.1 Can_InitController

CAN229:

Service name:	Can_InitController	

Syntax:	`void Can_InitController(`
	`    uint8 Controller,`
	`    const Can_ControllerConfigType* Config`
	`)`	

Service ID[hex]:	0x02	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	Controller	CAN controller to be initialized
	Config	Pointer to controller configuration.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	This function initializes only CAN controller specific settings.	

The function Can_InitController 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 7.3

The CAN controller must be in state CANIF_CS_STOPPED when this function is called (see CAN256 and CAN260).

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

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

CAN192: The function Can_InitController shall raise the error CAN_E_TIMEOUT if the initialization could not be performed (indicates defective hardware).

CAN187: If development error detection for the Can module is enabled: The function Can_InitController shall raise the error CAN_E_UNINIT if the driver is not yet initialized.

CAN188: If development error detection for the Can module is enabled: The function Can_InitController shall raise the error CAN_E_PARAM_POINTER if the parameter Config is an null pointer.

CAN189: If development error detection for the Can module is enabled: The function Can_InitController shall raise the error CAN_E_PARAM_CONTROLLER if the parameter `Controller` is out of range.

CAN190: If development error detection for the Can module is enabled: if the controller is not in state CANIF_CS_STOPPED , the function Can_InitController shall raise the error CAN_E_TRANSITION.

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	--
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Can_ReturnType	CAN_OK: transition initiated CAN_NOT_OK: development or production or a wakeup during transition to 'sleep' occured
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]

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.

For all state changes except the change to state CANIF_CS_SLEEP, the function does not wait until the state change has really performed. Anyway, this function is asynchronous because the actual result may occur later. However, neither callback nor notification will report the actual state change afterwards.

CAN196: The function Can_SetControllerMode shall enable interrupts that are needed in the new state. 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. Disabling of CAN interrupts shall not be executed, when CAN interrupts have been disabled by function CAN_DisableControllerInterrupts.

CAN201: The function Can_SetControllerMode shall raise the error CAN_E_TIMEOUT if the initialization could not be performed (indicates defective hardware, not for sleep transition).

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.	

CAN049: The function Can_DisableControllerInterrupts shall disable all interrupts for this CAN controller only at the first call of this function.

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.

CAN292: The function Can_DisableControllerInterrupts shall raise the production error CAN_E_TIMEOUT if the disabling of the interrupts could not be performed (indicates defective hardware).

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.	

CAN050: The function Can_EnableControllerInterrupts shall enable all interrupts that must be enabled according the current software status.

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.

CAN293: The function Can_EnableControllerInterrupts shall raise the production error CAN_E_TIMEOUT if the enabling of the interrupts could not be performed (indicates defective hardware).

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_Cbk_CheckWakeup

CAN360:

Service name:	Can_Cbk_CheckWakeup	

Syntax:	`Std_ReturnType Can_Cbk_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:	Std_ReturnType	E_OK: A wakeup was detected for the given controller. E_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_Cbk_CheckWakeup shall check if the requested CAN controller has detected a wakeup. If a wakeup event was successfully detected, the function shall return E_OK, otherwise E_NOT_OK.

CAN362: If development error detection for the Can module is enabled: The function Can_Cbk_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_Cbk_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(`
	`    uint8 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 occured CAN_BUSY: No TX hardware buffer available or preemptive call of Can_Write that can't be implemented reentrant
Description:	--	

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

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

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 or identical priority than that for the current request:
- The transmission of the L-PDU with lower or identical 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

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).

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 if the driver 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 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 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 if the parameter `PduInfo` or the SDU pointer inside PduInfo is a null-pointer.

8.4 Call-back notifications

The Can module does not provide callback functions.
Only synchronous MCAL API may be used for external CAN controllers.

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.

CAN110: There is no requirement regarding the execution order of the CAN main processing functions.

8.5.1.1 Can_MainFunction_Write

CAN225:

Service name:	Can_MainFunction_Write

Syntax:	`void Can_MainFunction_Write(`
	
	 `)`

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.

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.

8.5.1.2 Can_MainFunction_Read

CAN226:

Service name:	Can_MainFunction_Read

Syntax:	`void Can_MainFunction_Read(`
	
	`)`

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.

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 not yet initialized.

8.5.1.3 Can_MainFunction_BusOff

CAN227:

Service name:	Can_MainFunction_BusOff

Syntax:	`void Can_MainFunction_BusOff(`
	
	`)`

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'.

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 not yet initialized.

8.5.1.4 Can_MainFunction_Wakeup

CAN228:

Service name:	Can_MainFunction_Wakeup

Syntax:	`void Can_MainFunction_Wakeup(`
	
	`)`

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'.

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 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***
Dem_ReportErrorStatus	Reports errors to the DEM.
CanIf_CheckValidation	--
CanIf_Cbk_CheckTransceiverWakeup	--
CanIf_Cbk_CheckControllerWakeup	--
CanIf_CancelTxConfirmation	--
CanIf_RxIndication	--
CanIf_ControllerBusOff	--
CanIf_TxConfirmation	--

Optional Interfaces

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

CAN235:

API function	***Description***
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.

8.6.2 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 complete sequences for Transmission, Reception and Error Handling.

9.2 Wakeup sequence

For Wakeup sequence diagrams refer to Specification of ECU State Manager [12].

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.

In the below given tables the configuration class per configuration parameter is specified. In fact, it is important to distinguish between the configuration-classes, because they will result in different implementations and design processes.

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*.

10.1.2 Variants

Variants describe sets of configuration parameters. E.g., VariantPC: only pre-compile time configuration parameters; VariantPB: 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.

10.2 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 configurator 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.

CAN047: The configuration data shall be human readable.

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.)

10.2.1 Variants

The Can module provides two variants of configuration sets:
CAN220: VariantPC: all variables are pre-compile time configurable
CAN221: VariantPB: (Mix of precompile and Post Build multiple selectable configurable configurations

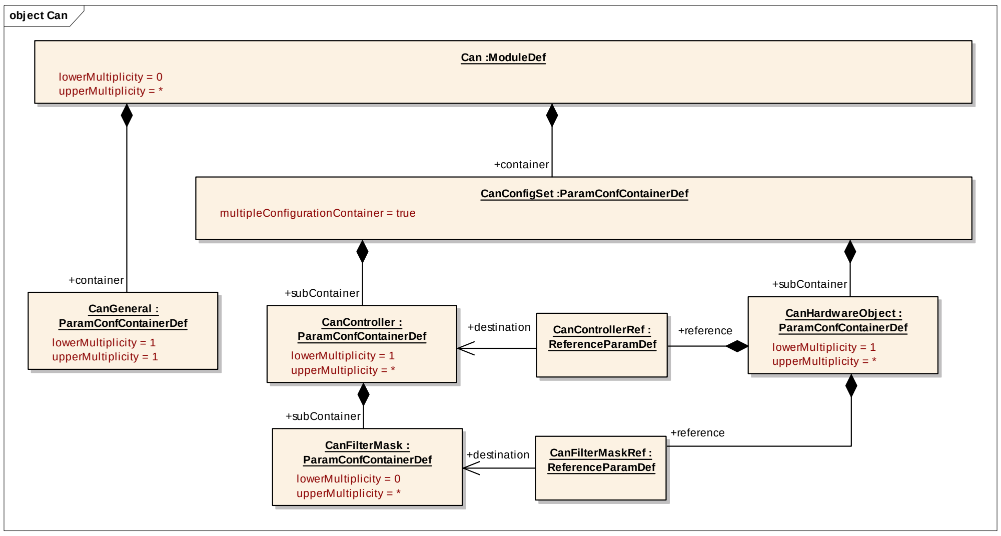


Figure 10-1: Can Module Configuration Layout

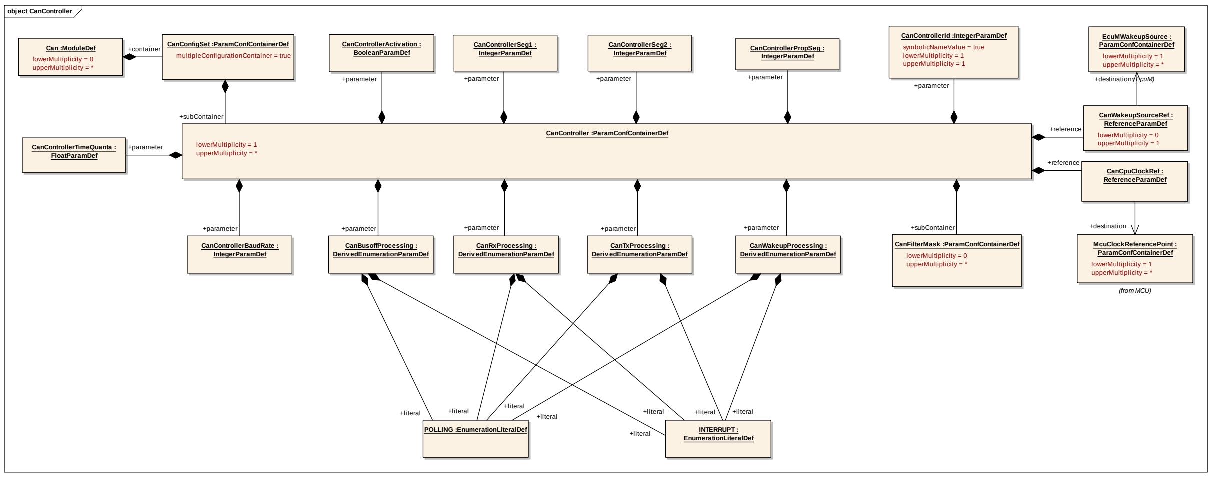

object CanController
Can :ModuleDef
lowerMultiplicity = 0
upperMultiplicity = *
+container
CanConfigSet :ParamConfContainerDef
multipleConfigurationContainer = true
CanControllerActivation : BooleanParamDef
CanControllerSeg1 : IntegerParamDef
CanControllerSeg2 : IntegerParamDef
CanControllerPropSeg : IntegerParamDef
CanControllerId :IntegerParamDef
symbolicNameValue = true
lowerMultiplicity = 1
upperMultiplicity = 1
EcuMWakeupSource : ParamConfContainerDef
lowerMultiplicity = 1
upperMultiplicity = *
+destination
(from EcuM)
+parameter
+subContainer
CanControllerTimeQuanta : FloatParamDef
CanController :ParamConfContainerDef
lowerMultiplicity = 1
upperMultiplicity = *
+reference
CanWakeupSourceRef : ReferenceParamDef
lowerMultiplicity = 0
upperMultiplicity = 1
CanCpuClockRef : ReferenceParamDef
+destination
CanControllerBaudRate : IntegerParamDef
CanBusoffProcessing : DerivedEnumerationParamDef
CanRxProcessing : DerivedEnumerationParamDef
CanTxProcessing : DerivedEnumerationParamDef
CanWakeupProcessing : DerivedEnumerationParamDef
CanFilterMask :ParamConfContainerDef
lowerMultiplicity = 0
upperMultiplicity = *
McuClockReferencePoint : ParamConfContainerDef
lowerMultiplicity = 1
upperMultiplicity = *
(from MCU)
+literal
POLLING :EnumerationLiteralDef
INTERRUPT : EnumerationLiteralDef

Figure 10-2: Can Controller Configuration Layout

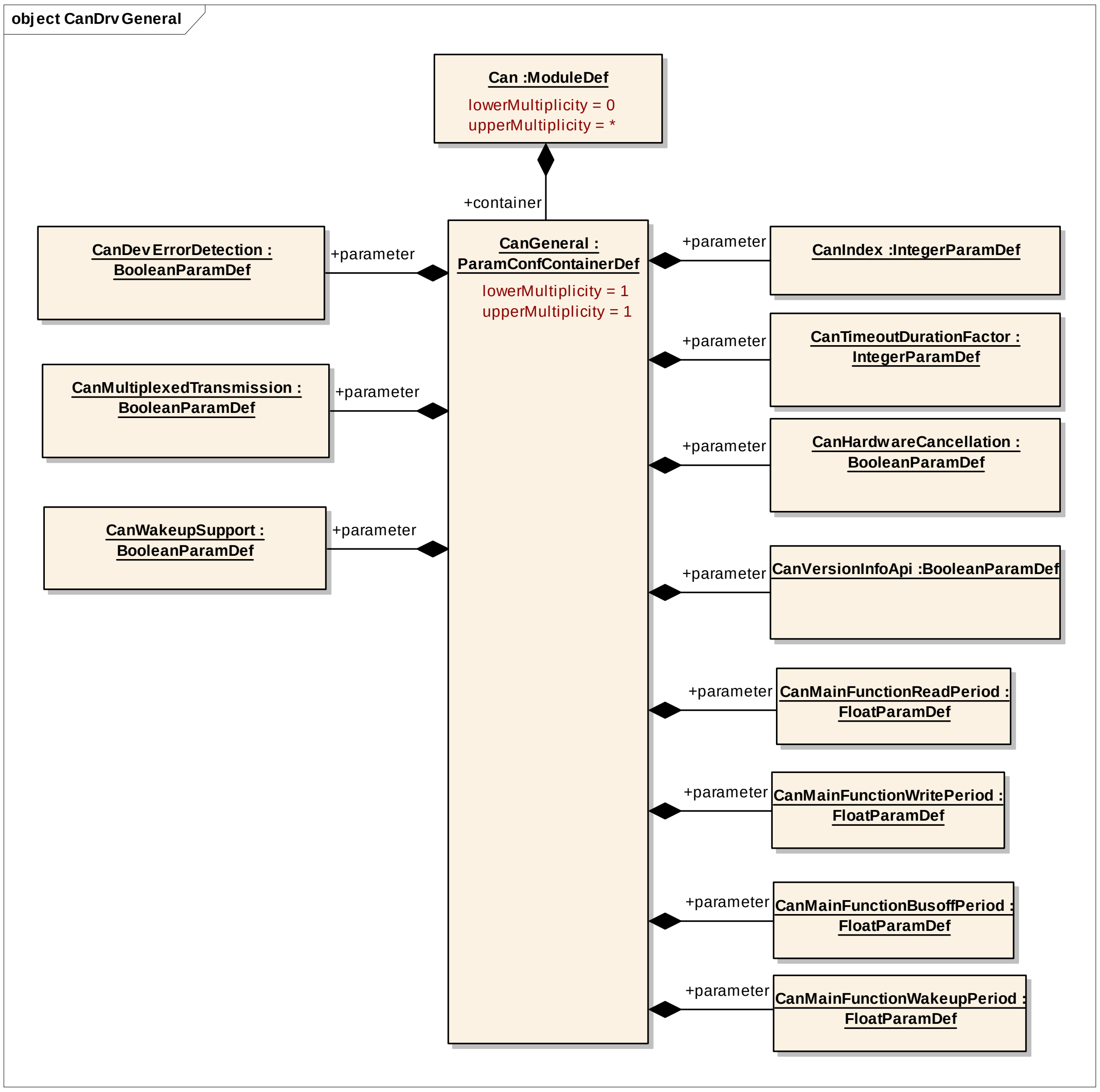

object CanDrvGeneral
Can :ModuleDef
lowerMultiplicity = 0
upperMultiplicity = *
+container
CanGeneral : ParamConfContainerDef
lowerMultiplicity = 1
upperMultiplicity = 1
CanDevErrorDetection : BooleanParamDef
CanMultiplexedTransmission : BooleanParamDef
CanWakeupSupport : BooleanParamDef
CanIndex :IntegerParamDef
CanTimeoutDurationFactor : IntegerParamDef
CanHardwareCancellation : BooleanParamDef
CanVersionInfoApi :BooleanParamDef
CanMainFunctionReadPeriod : FloatParamDef
CanMainFunctionWritePeriod : FloatParamDef
CanMainFunctionBusoffPeriod : FloatParamDef
CanMainFunctionWakeupPeriod : FloatParamDef
+parameter

Figure 10-3: Can General Configuration Layout

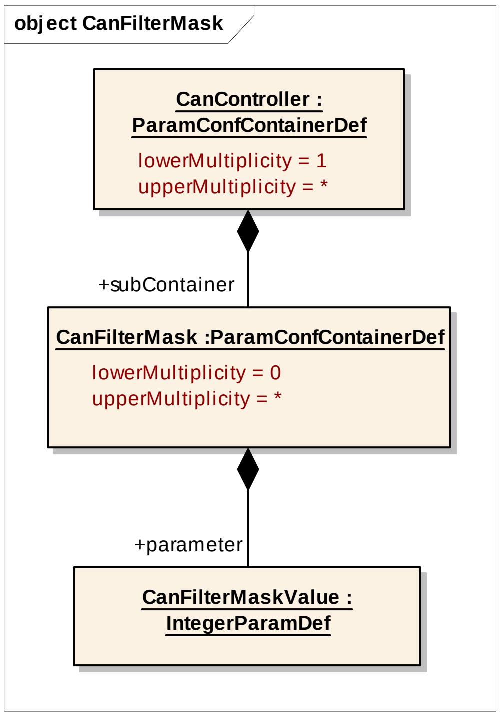


Figure 10-4: Can Filter Mask Configuration Layout

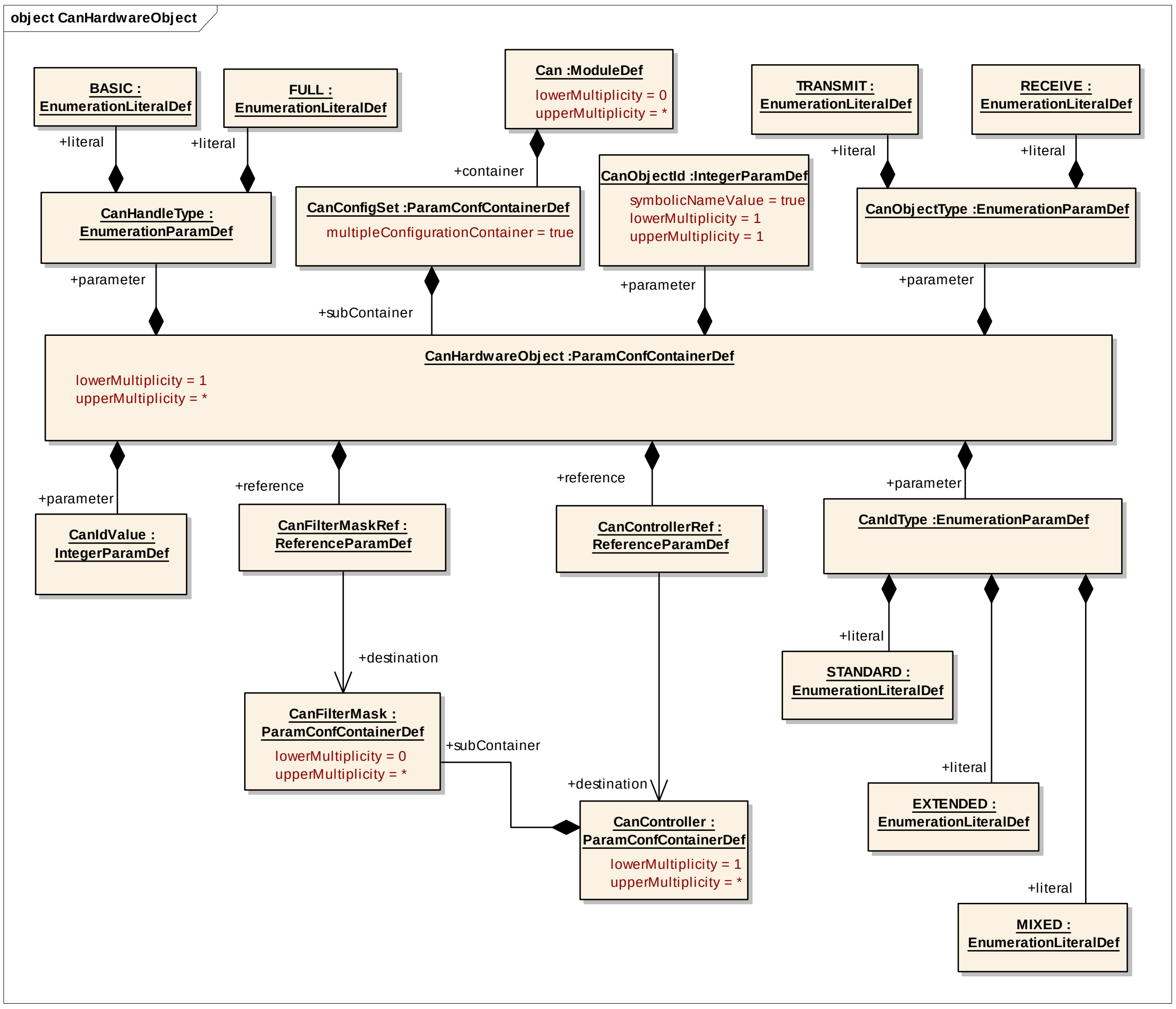

object CanHardwareObject
BASIC : EnumerationLiteralDef
FULL : EnumerationLiteralDef
Can :ModuleDef
lowerMultiplicity = 0
upperMultiplicity = *
TRANSMIT : EnumerationLiteralDef
RECEIVE : EnumerationLiteralDef
+literal
+literal
+container
CanHandleType : EnumerationParamDef
CanConfigSet :ParamConfContainerDef
multipleConfigurationContainer = true
CanObjectId :IntegerParamDef
symbolicNameValue = true
lowerMultiplicity = 1
upperMultiplicity = 1
CanObjectType :EnumerationParamDef
+parameter
+subContainer
CanHardwareObject :ParamConfContainerDef
lowerMultiplicity = 1
upperMultiplicity = *
+reference
+destination
CanIdValue : IntegerParamDef
CanFilterMaskRef : ReferenceParamDef
CanControllerRef : ReferenceParamDef
CanIdType :EnumerationParamDef
STANDARD : EnumerationLiteralDef
CanFilterMask : ParamConfContainerDef
lowerMultiplicity = 0
upperMultiplicity = *
EXTENDED : EnumerationLiteralDef
CanController : ParamConfContainerDef
lowerMultiplicity = 1
upperMultiplicity = *
MIXED : EnumerationLiteralDef

Figure 10-5: 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 :**
Container Name	CanGeneral{CanDriverGeneralConfiguration}

Description	This container contains the parameters related each CAN Driver Unit.		
Configuration Parameters			

SWS Item	**CAN064 :**		
Name	CanDevErrorDetection {CAN_DEV_ERROR_DETECT}		
Description	Switches the Development Error Detection and Notification ON or OFF.		
Multiplicity	1		
Type	BooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Can module		

SWS Item	**CAN069 :**		
Name	CanHardwareCancellation {CAN_HW_TRANSMIT_CANCELLATION}		
Description	Specifies if hardware cancellation shall be supported.ON or OFF		
Multiplicity	1		
Type	BooleanParamDef		
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	**CAN320 :**		
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	IntegerParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**CAN355 :**		
Name	CanMainFunctionBusoffPeriod		
Description	This parameter describes the period for cyclic call to Can_MainFunction_Busoff. Unit is seconds.		
Multiplicity	1		
Type	FloatParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**CAN356 :**
Name	CanMainFunctionReadPeriod
Description	This parameter describes the period for cyclic call to Can_MainFunction_Read. Unit is seconds.
Multiplicity	1

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

SWS Item	**CAN357 :**		
Name	CanMainFunctionWakeupPeriod		
Description	This parameter describes the period for cyclic call to Can_MainFunction_Wakeup. Unit is seconds.		
Multiplicity	1		
Type	FloatParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**CAN358 :**		
Name	CanMainFunctionWritePeriod		
Description	This parameter describes the period for cyclic call to Can_MainFunction_Write. Unit is seconds.		
Multiplicity	1		
Type	FloatParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**CAN095 :**		
Name	CanMultiplexedTransmission {CAN_MULTIPLEXED_TRANSMISSION}		
Description	Specifies if multiplexed transmission shall be supported.ON or OFF		
Multiplicity	1		
Type	BooleanParamDef		
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 :**		
Name	CanTimeoutDurationFactor {CAN_TIMEOUT_DURATION}		
Description	Specifies the maximum number of loops for blocking function until a timeout is raised in short term wait loops.		
Multiplicity	1		
Type	IntegerParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Can module		

SWS Item	**CAN106 :**		
Name	CanVersionInfoApi {CAN_VERSION_INFO_API}		
Description	Switches the Can_GetVersionInfo() API ON or OFF.		
Multiplicity	1		
Type	BooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Can module		

SWS Item	**CAN330 :**		
Name	CanWakeupSupport {CAN_WAKEUP_SUPPORT}		
Description	CAN driver support for wakeup over CAN Bus.		
Multiplicity	1		
Type	BooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Can module dependency: CAN Hardware Unit supports wakeup over CAN		

No Included Containers

10.2.4 CanController

SWS Item	**CAN354 :**
Container Name	CanController{CanController}
Description	This container contains the configuration parameters of the CAN controller(s).
Configuration Parameters	

SWS Item	**CAN314 :**		
Name	CanBusoffProcessing {CAN_BUSOFF_PROCESSING}		
Description	Enables / disables API Can_MainFunction_BusOff() for handling busoff events in polling mode.		
Multiplicity	1		
Type	EnumerationParamDef		
Range	INTERRUPT	Interrupt Mode of operation.	
	POLLING	Polling Mode of operation.	
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Can module, CanIf module dependency: CANIF_POLLING_BUSOFF		

SWS Item	**CAN315 :**		
Name	CanControllerActivation {CAN_CONTROLLER_ACTIVATION}		
Description	Defines if a CAN controller is used in the configuration.		
Multiplicity	1		
Type	BooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants

	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Can module		

SWS Item	**CAN005 :**		
Name	CanControllerBaudRate {CAN_CONTROLLER_BAUD_RATE}		
Description	Specifies the buadrate of the controller in kbps.		
Multiplicity	1		
Type	IntegerParamDef		
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	**CAN316 :**		
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	IntegerParamDef (Symbolic Name generated for this parameter)		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**CAN073 :**		
Name	CanControllerPropSeg {CAN_CONTROLLER_PROP_SEG}		
Description	Specifies propagation delay in time quantas.		
Multiplicity	1		
Type	IntegerParamDef		
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 :**		
Name	CanControllerSeg1 {CAN_CONTROLLER_PHASE_SEG1}		
Description	Specifies phase segment 1 in time quantas.		
Multiplicity	1		
Type	IntegerParamDef		
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 :**
Name	CanControllerSeg2 {CAN_CONTROLLER_PHASE_SEG2}
Description	Specifies phase segment 2 in time quantas.
Multiplicity	1
Type	IntegerParamDef

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	**CAN063 :**		
Name	CanControllerTimeQuanta {CAN_CONTROLLER_TIME_QUANTA}		
Description	Specifies the time quanta for the controller. The calculation of the resulting prescaler value depending on module clocking and time quanta shall be done offline Hardware specific.		
Multiplicity	1		
Type	FloatParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency			

SWS Item	**CAN317 :**		
Name	CanRxProcessing {CAN_RX_PROCESSING}		
Description	Enables / disables API Can_MainFunction_Read() for handling PDU reception events in polling mode.		
Multiplicity	1		
Type	EnumerationParamDef		
Range	INTERRUPT	Interrupt Mode of operation.	
	POLLING	Polling Mode of operation.	
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Can module, CanIf module dependency: CANIF_POLLING_RECEIVE		

SWS Item	**CAN318 :**		
Name	CanTxProcessing {CAN_TX_PROCESSING}		
Description	Enables / disables API Can_MainFunction_Write() for handling PDU transmission events in polling mode.		
Multiplicity	1		
Type	EnumerationParamDef		
Range	INTERRUPT	Interrupt Mode of operation.	
	POLLING	Polling Mode of operation.	
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Can module, CanIf module dependency: CANIF_POLLING_TRANSMIT		

SWS Item	**CAN319 :**	
Name	CanWakeupProcessing {CAN_WAKEUP_PROCESSING}	
Description	Enables / disables API Can_MainFunction_Wakeup() for handling wakeup events in polling mode.	
Multiplicity	1	
Type	EnumerationParamDef	
Range	INTERRUPT	Interrupt Mode of operation.
	POLLING	Polling Mode of operation.

ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: Can module, CanIf module dependency: CANIF_POLLING_WAKEUP		

SWS Item	**CAN313 :**		
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 :**		
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***
CanFilterMask	0..*	This container contains the configuration (parameters) of the CAN Filter Mask(s).

10.2.5 CanHardwareObject

SWS Item	**CAN324 :**
Container Name	CanHardwareObject{CanHardwareObject}
Description	This container contains the configuration (parameters) of CAN Hardware Objects.
Configuration Parameters	

SWS Item	**CAN324 :**		
Name	CanHandleType {CAN_HANDLE_TYPE}		
Description	Specifies the type (Full-CAN or Basic-CAN) of a hardware object.		
Multiplicity	1		
Type	EnumerationParamDef		
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 :**		
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	EnumerationParamDef		
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 :**		
Name	CanIdValue {CAN_ID_VALUE}		
Description	Specifies (together with the filter mask) the identifiers range that passes the hardware filter.		
Multiplicity	1		
Type	IntegerParamDef		
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 :**		
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	IntegerParamDef (Symbolic Name generated for this parameter)		
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 :**	
Name	CanObjectType {CAN_OBJECT_TYPE}	
Description	Specifies if the HardwareObject is used as Transmit or as Receive object	
Multiplicity	1	
Type	EnumerationParamDef	
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 :**		
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 :**		
Name	CanFilterMaskRef {CAN_MASK_REFERENCE}		
Description	Reference to the filter mask that is used for hardware filtering togerther with the CAN_ID_VALUE		
Multiplicity	1		
Type	Reference to CanFilterMask		
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 CanFilterMask

SWS Item	**CAN351 :**
Container Name	CanFilterMask{CanFilterMask}
Description	This container contains the configuration (parameters) of the CAN Filter Mask(s).
Configuration Parameters	

SWS Item	**CAN066 :**		
Name	CanFilterMaskValue {CAN_FILTER_MASK_VALUE}		
Description	Describes a mask for hardware-based filtering of CAN identifiers It shall be distinguished between - Standard identifier mask - Extended identifier mask.		
Multiplicity	1		
Type	IntegerParamDef		
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 configuraton for optimization of the SW filters.		

No Included Containers

10.2.7 CanConfigSet

SWS Item	**CAN343 :**
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.

10.3 Published Information

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

The standard common published information like

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

is provided in the BSW Module Description Template (see [13] Figure 4.1 and Figure 7.1).

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

11 Changes to Release 2.1

11.1 Deleted SWS Items

SWS Item	***Rationale***
CAN057	No requirement: ID removed, text kept
CAN038	No requirement: ID removed, text kept
CAN090	No requirement: ID removed, text kept
CAN173	Redundant requirement removed, requirement is already described in CAN176
CAN102	Redundant requirement removed, requirement is already described in CAN234
CAN193	Redundant requirement removed, requirement is already described in CAN048

11.2 Replaced SWS Items

SWS Item of Release 1	***replaced by SWS Item***	***Rationale***
CAN097	CAN285, CAN286, CAN287, CAN288	Made requirement atomic
CAN067	CAN324	Gave new ID, because CAN067 was already in use.
CAN067	CAN325	Gave new ID, because CAN067 was already in use.

11.3 Changed SWS Items

SWS Item	***Rationale***
CAN225	Function changed to scheduled function
CAN226	Function changed to scheduled function
CAN227	Function changed to scheduled function
CAN228	Function changed to scheduled function
CAN325	Limitation to Rx objects removed

11.4 Added SWS Items

SWS Item	***Rationale***
CAN280	Gave ID to existing requirement
CAN281	Gave ID to existing requirement
CAN282	Gave ID to existing requirement
CAN283	Gave ID to existing requirement
CAN284	Gave ID to existing requirement
CAN290	Gave ID to existing requirement
CAN291	Gave ID to existing requirement
CAN292	Requirement for the function Can_DisableControllerInterrupts
CAN293	Requirement for the function Can_EnableControllerInterrupts
CAN294	Gave ID to existing requirement

CAN295	Gave ID to existing requirement
CAN296	Gave ID to existing requirement
CAN297	Gave ID to existing requirement
CAN298	Gave ID to existing requirement
CAN299	Gave ID to existing requirement
CAN300	Gave ID to existing requirement
CAN301	Gave ID to existing requirement
CAN355	Addition of parameter for cyclic call of scheduled function
CAN356	Addition of parameter for cyclic call of scheduled function
CAN357	Addition of parameter for cyclic call of scheduled function
CAN358	Addition of parameter for cyclic call of scheduled function
CAN359	Addition of parameter for wakeup source id
CAN360	UML model linking of Can_Cbk_CheckWakeup
CAN361	Requirement for the function Can_Cbk_CheckWakeup
CAN362	Requirement for the function Can_Cbk_CheckWakeup
CAN363	Requirement for the function Can_Cbk_CheckWakeup
CAN364	Requirement for wakeup ISR

12 Changes during SWS Improvements by Technical Office

12.1 Deleted SWS Items

SWS Item	***Rationale***
CAN001	No requirement: ID removed, text kept
CAN003	No requirement: ID removed, text kept

12.2 Replaced SWS Items

SWS Item of Release 1	***replaced by SWS Item***	***Rationale***
CAN008	CAN173, CAN174, CAN176	Made requirement atomic
CAN015	CAN212, CAN213, CAN214, CAN215, CAN216, CAN217, CAN218, CAN219	Made requirement atomic
CAN046	CAN238, CAN239, CAN240, CAN241	Made requirement atomic
CAN094	CAN242, CAN243, CAN244	Made requirement atomic
CAN052	CAN257, CAN258	Made requirement atomic
CAN114	CAN277, CAN278	Made requirement atomic
CAN101	CAN402, CAN403	Made requirement atomic and improved description of Multiplexed Transmission.

12.3 Changed SWS Items

Many requirements have been changed to improve understandability without changing the technical contents.

SWS Item	***Rationale***
CAN215, CAN286	Improve description of Cancel Transmit functionality
CAN076	Improve description of Multiplexed Transmission
CAN034	SchM_Can.h added to support requirement BSW00435.
CAN271	API CanIf_SetWakeupEvent replaced by EcuM_CheckWakeup to be compliant to wakeup concept.

12.4 Added SWS Items

SWS Item	***Rationale***
CAN177	Requirement for the function Can_GetVersionInfo
CAN178	Requirement for the function Can_MainFunction_Write
CAN179	Requirement for the function Can_MainFunction_Write
CAN180	Requirement for the function Can_MainFunction_Read
CAN181	Requirement for the function Can_MainFunction_Read

CAN183	Requirement for the function Can_MainFunction_BusOff
CAN184	Requirement for the function Can_MainFunction_BusOff
CAN185	Requirement for the function Can_MainFunction_Wakeup
CAN186	Requirement for the function Can_MainFunction_Wakeup
CAN187	Requirement for the function Can_InitController
CAN188	Requirement for the function Can_InitController
CAN189	Requirement for the function Can_InitController
CAN190	Requirement for the function Can_InitController
CAN192	Requirement for the function Can_InitController
CAN193	Requirement for the function Can_SetControllerMode
CAN196	Requirement for the function Can_SetControllerMode
CAN197	Requirement for the function Can_SetControllerMode
CAN198	Requirement for the function Can_SetControllerMode
CAN199	Requirement for the function Can_SetControllerMode
CAN200	Requirement for the function Can_SetControllerMode
CAN201	Requirement for the function Can_SetControllerMode
CAN202	Requirement for the function Can_DisableControllerInterrupts
CAN204	Requirement for the function Can_DisableControllerInterrupts
CAN205	Requirement for the function Can_DisableControllerInterrupts
CAN206	Requirement for the function Can_DisableControllerInterrupts
CAN208	Requirement for the function Can_EnableControllerInterrupts
CAN209	Requirement for the function Can_EnableControllerInterrupts
CAN210	Requirement for the function Can_EnableControllerInterrupts
CAN220	Each variant gets an individual requirement ID
CAN221	Each variant gets an individual requirement ID
CAN222	UML model linking of imported types
CAN223	UML model linking of Can_Init
CAN224	UML model linking of Can_GetVersionInfo
CAN225	UML model linking of Can_MainFunction_Write
CAN226	UML model linking of Can_MainFunction_Read
CAN227	UML model linking of Can_MainFunction_BusOff
CAN228	UML model linking of Can_MainFunction_Wakeup
CAN229	UML model linking of Can_InitController
CAN230	UML model linking of Can_SetControllerMode
CAN231	UML model linking of Can_DisableControllerInterrupts
CAN232	UML model linking of Can_EnableControllerInterrupts
CAN233	UML model linking of Can_Write
CAN234	UML model linking of mandatory interfaces
CAN235	UML model linking of optional interfaces
CAN237	Gave ID to existing requirement
CAN245	Gave ID to existing requirement
CAN246	Gave ID to existing requirement
CAN247	Gave ID to existing requirement
CAN248	Gave ID to existing requirement
CAN249	Gave ID to existing requirement
CAN250	Gave ID to existing requirement
CAN251	Gave ID to existing requirement
CAN252	Gave ID to existing requirement
CAN255	Gave ID to existing requirement
CAN256	Gave ID to existing requirement
CAN259	Gave ID to existing requirement
CAN260	Gave ID to existing requirement
CAN261	Gave ID to existing requirement
CAN262	Gave ID to existing requirement
CAN263	Gave ID to existing requirement
CAN264	Gave ID to existing requirement
CAN265	Gave ID to existing requirement
CAN266	Gave ID to existing requirement

CAN267	Gave ID to existing requirement
CAN268	Gave ID to existing requirement
CAN269	Gave ID to existing requirement
CAN270	Gave ID to existing requirement
CAN271	Gave ID to existing requirement
CAN272	Gave ID to existing requirement
CAN273	Gave ID to existing requirement
CAN274	Gave ID to existing requirement
CAN275	Gave ID to existing requirement
CAN276	Gave ID to existing requirement
CAN279	Gave ID to existing requirement
CAN399, CAN400	Improve description of Cancel Transmit functionality
CAN401	Improve description of Multiplexed Transmission
CAN404	Added to support requirement BSW00435