Document Title	Specification of CAN Driver
Document Owner	AUTOSAR GbR
Document Responsibility	AUTOSAR GbR
Document Version	2.0.1
Document Status	Final

Document Change History

Date	Version	Changed by	Change Description
27.06.2006	2.0.1	AUTOSAR Administration	Layout Adaptations
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 describes the functionality and API for the CAN driver.

CAN001: The CAN driver 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 driver is the CAN interface (see also [BSW12092]).

The CAN driver provides services for initiating transmissions and calls the callback functions of the CAN Interface 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.

CAN003: Several CAN controllers can be controlled by the CAN driver as long as they belong to the same CAN Hardware Unit.

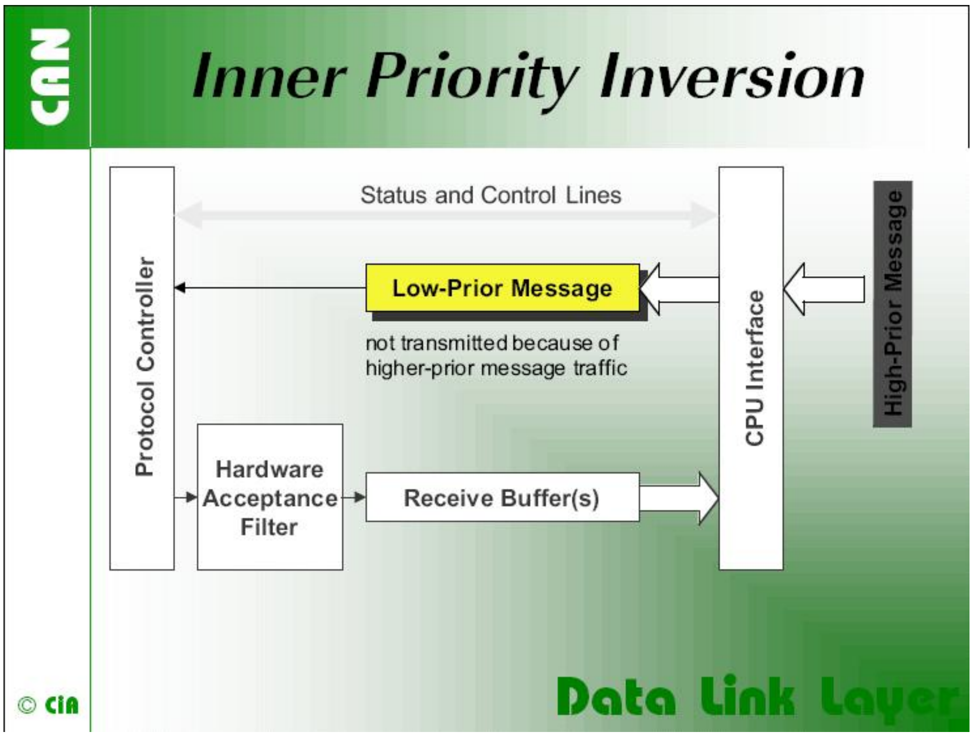
For a closer description of CAN controller and CAN Hardware Unit see chapter Acronyms and abbreviations and a diagram in [5].

2 Acronyms and abbreviations

Acronyms and abbreviations, which have a local scope and therefore are not contained in the AUTOSAR glossary, are described in this chapter.

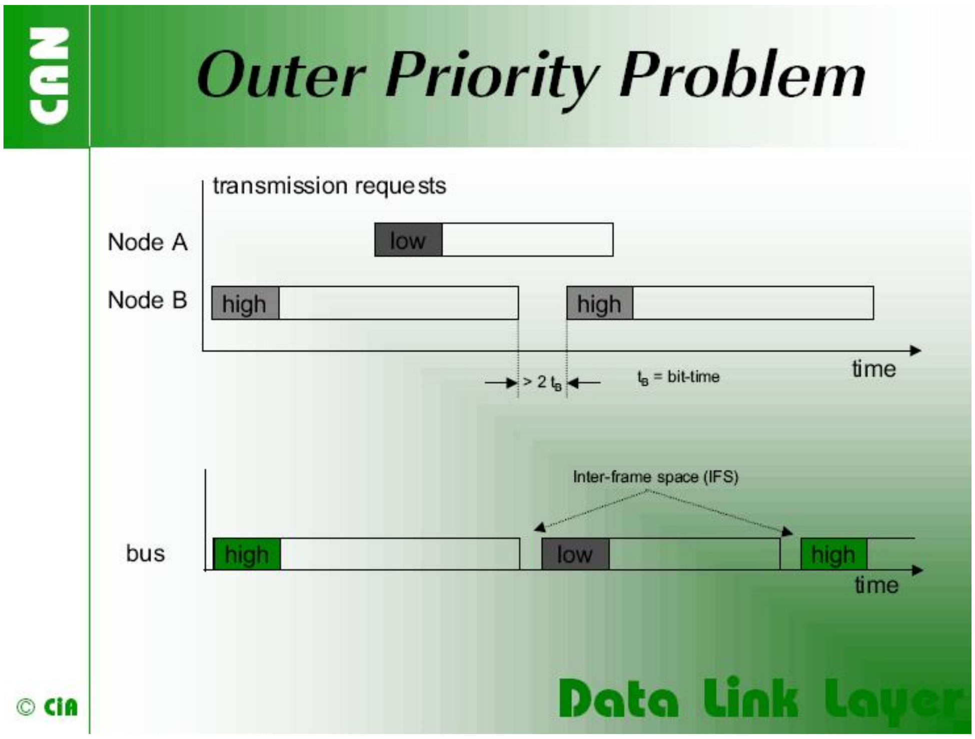
Abbreviation / Acronym:	***Description:***
CAN controller	A CAN controller serves exactly one physical channel.
CAN Hardware Unit	A CAN Hardware Unit may consists of one or multiple CAN controllers of the same type and one or multiple CAN RAM areas. The CAN Hardware Unit is either on-chip, or an external device. The CAN Hardware Unit is represented by one CAN driver.
Hardware Object	A Hardware Object is defined as L-PDU buffer inside the CAN RAM of the CAN Hardware Unit.
Hardware Rx Handle	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 Tx Handle	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.
L-PDU	Data Link Layer Protocol Data Unit. Consists of Identifier, DLC and Data (SDU). (see [13])
L-PDU Handle	The L-PDU handle is defined and placed inside the CAN interface layer. Typically each handle represents a L-PDU or a range of PDUs, and is a constant structure with information for Tx/Rx processing.
L-SDU	Data Link Layer Service Data Unit. Data that is transported inside the L-PDU. (see [13])
Outer Priority Inversion	Occurs when a time gap is between two consecutive TX L-PDU. In this case a lower priority L-PDU from another node can prevent sending the next L-PDU because the higher priority L-PDU can't participate in the bus access because it comes too late.
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. It consists of an RX and TX line including the necessary hardware drivers (see 2.2)
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.
DLC	Data Length Code (part of L-PDU that describes the SDU length)
HRH	Hardware Receive Handle
HTH	Hardware Transmit Handle
ISR	Interrupt Service Routine
PDU	Protocol Data Unit
SDU	Service Data Unit
SFR	Special Function Register. Hardware register that controls the controller behavior.

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)



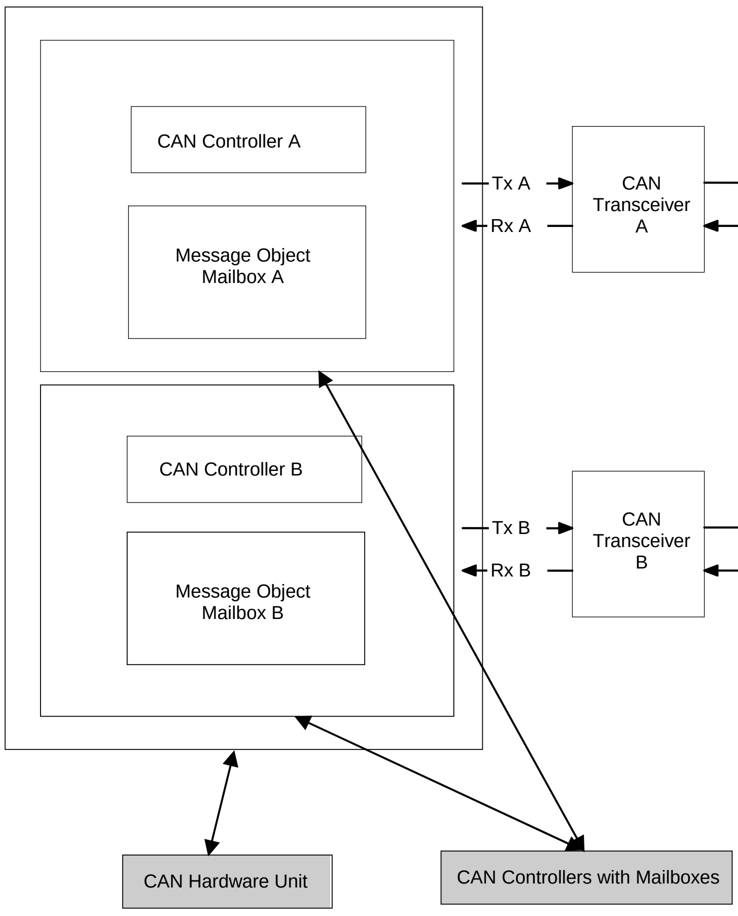
"The problem of outer priority inversion may occur in some CAN implementations. Let us assume that a CAN node wishes to transmit a package of consecutive messages with high priority, which are stored in different message buffers. If the interframe space between these messages on the CAN network is longer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

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:



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_CAN_Interface.pdf

[6] Specification of Development Error Tracer
AUTOSAR_SWS_DevelopmentErrorTracer.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 Module Operating System
AUTOSAR_SWS_OS.pdf

[10] Specification of ECU Configuration
AUTOSAR_ECU_Configuration.pdf

[11] Programming Coding Guidelines
AUTOSAR_ProgrammingCodingGuidelines.pdf

3.2 Related standards and norms

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

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

[14] 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 CAN interface 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 driver.

CAN Remote Frames are not supported by the AUTOSAR CAN driver. Received remote frames are not further processed.

4.2 Driver scope

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

CAN077: For CAN Hardware Units of different type different CAN drivers need to be implemented.
In case several CAN Hardware Units (of same or different vendor) are implemented in one ECU the function names, and global variables must be modified such that no two functions with the same name are generated.

The name can be extended with a Vendor ID (in case of several CAN Hardware Units from different vendors) and a Type ID (in case if several different Hardware Units from same Vendor). Any combination of these extensions is possible.
The CAN interface is responsible for calling the correct function. The necessary information shall be given in an XML file during configuration.

See [5] for description how several CAN drivers are handled by the CAN interface.

4.3 Applicability to car domains

The CAN driver Layer can be used for any application, where the CAN protocol is used.

4.4 Applicability to safety related environments

The CAN driver layer is not applicable for safety relevant systems.

5 Dependencies to other modules

This section describes the relations to other modules within the basic software.

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

CAN046: The CAN driver is (with exceptions mentioned below) not allowed to use any service of other drivers if the CAN controller is on-chip. All on-chip hardware resources that are used by the CAN controller must be initialized by calling Can_Init.
The only exception to this is the digital I/O pin configuration, which is done by the port driver.
Register settings that are 'shared' with other modules are configured by the MCU driver (SPAL see [8]). The MCU initialization must be done before the CAN driver is initialized.

CAN094: If an off-chip CAN controller is used[3], the driver uses services of other MCAL drivers (i.e. SPI). These drivers need to be up and running before the CAN controller can be initialized. The sequence of initialization of different drivers is partly specified in [7]. Only Synchronous APIs may be used because the CAN driver does not provide callback functions that can be called by the MCAL driver. Thus the type of connection between µC and CAN Device has only impact on implementation and not on the API.

5.1.3 System Services

In special hardware cases the CAN driver must poll for events of the hardware.
That must be done with a timeout in case the hardware doesn't react in the expected time (hardware error) to prevent endless loops. As long as the system service does not provide a free running timer this timeout is realized with a fixed number of loops.[4]

Reason: The blocking time of the CAN driver function that is waiting for hardware reaction shall be shorter than the scheduled main function (i.e. Can_MainFunction_Read) trigger period, so the a scheduled main function can't be used for that purpose.

In case consistency concepts (resources/critical sections) are offered by the AUTOSAR OS, the according OS services will be used by the CAN driver.

[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.
[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.

5.1.4 CAN Driver Users

CAN058: The CAN driver only communicates with the CAN interface 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 CAN interface 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 files named:
- Can_Lcfg.c – for link time configurable parameters and
- Can_PBcfg.c – for post build time configurable parameters.

These files shall contain all link time and post-build time configurable parameters.

5.2.2 Header file structure

CAN034:

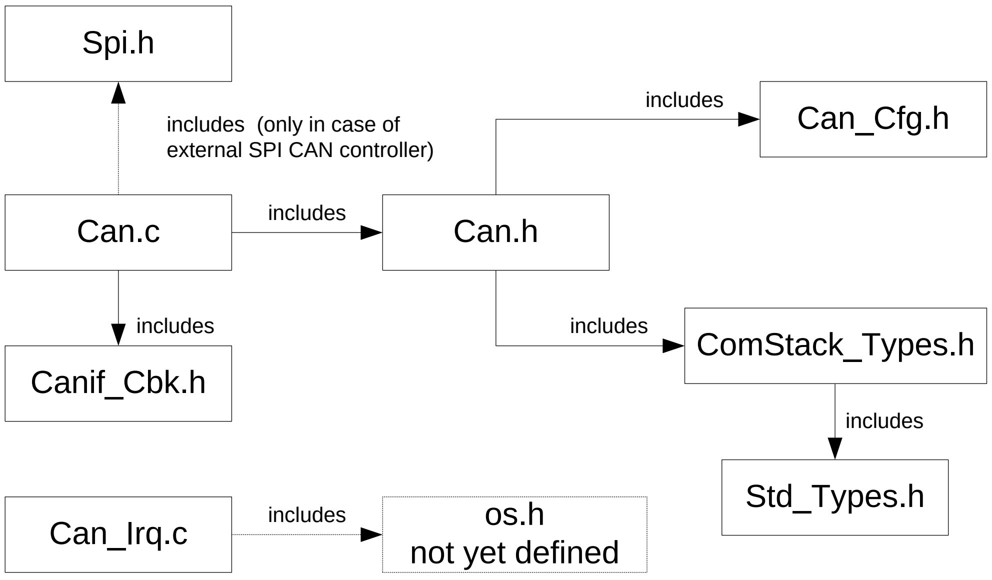


Figure 5-1: File structure for the CAN driver

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 driver does not provide callback functions (no Can_Cbk.h, see also CAN094.

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

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

6 Requirements traceability

Document: General requirements on Basic Software Modules [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	CAN102
[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

Requirement	***Satisfied by***
[BSW101] Initialization interface	CAN008
[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

Requirement	***Satisfied by***
[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	CAN001
[BSW162] ECU layout abstraction	Not applicable (done in CAN interface)
[BSW00324] Do not use HIS Library	Fulfilled by the concept of CAN driver and CAN interface
[BSW005] No hard coded horizontal interfaces within MCAL	CAN046
[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 driver, 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 CAN interface 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 driver)
[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

Requirement	***Satisfied by***
[BSW00370] Separation of callback interface from API	CAN036
[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 driver that controls multiple channels
[BSW00312] Shared code shall be reentrant	CAN038, CAN090
[BSW006] Platform independency	CAN001
[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	Not 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	CAN008
[BSW00414] Parameter of init function	CAN008
[BSW00376] Return type and parameters of main processing functions	CAN031
[BSW00359] Return type of callback functions	Not applicable (no callback functions implemented in CAN driver)
[BSW00360] Parameters of callback functions	No callbacks implemented in CAN driver
[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
[BSW009], [BSW00401], [BSW172], [BSW010], [BSW00333], [BSW00374], [BSW00379], [BSW003], [BSW00318], [BSW00321], [BSW00341], [BSW00334]	Software Documentation Requirements are not covered in the CAN driver SWS

Document: General requirements on SPAL [3]

Requirement	***Satisfied by***
[BSW12263] Object code compatible configuration concept	CAN021
[BSW12056] Configuration of notification mechanisms	CAN102
[BSW12267] Configuration of wake-up sources	CAN052, CAN018
[BSW12057] Driver module initialization	CAN008
[BSW12125] Initialization of hardware resources	CAN053

Requirement	***Satisfied by***
[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	CAN057, CAN028
[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 (state machine defined in a way that degradation of running operation does occur)
[BSW12448] Behavior after development error detection	done in chapter 7.8.1
[BSW12067] Setting of wake-up conditions	CAN052, 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	Not applicable (this rqmt. does not affect the HIS CAN driver)

Document: Requirements on 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	CAN001
[BSW01035] Multiple CAN controller support	CAN003
[BSW01036] CAN Identifier Length Configuration	CAN065

Requirement	***Satisfied by***
[BSW01037] Hardware Filter Configuration	CAN066, CAN067
[BSW01038] Bit Timing Configuration	CAN005, CAN063, CAN073, CAN074, CAN075
[BSW01039] CAN Hardware Object Handle definitions	CAN068
[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 driver Module Initialization	CAN008
[BSW01042] Selection of static configuration sets	CAN008, CAN062
[BSW01043] Enable/disable Interrupts	CAN049, CAN050
[BSW01059] Data Consistency	CAN011, CAN012
[BSW01045] Reception Indication Service	CAN013
[BSW01049] Dynamic transmission request service	CAN015
[BSW01051] Transmit Confirmation	CAN016
[BSW01053] CAN controller mode select	CAN015, CAN017
[BSW01054] Wake-up Notification	CAN018
[BSW01132] Mixed mode for notification detection on CAN HW	CAN099
[BSW01133] HW Transmit Cancellation Support	CAN097, CAN098
[BSW01134] Multiplexed Transmission	CAN101, 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 driver writes the L-PDU in an appropriate buffer inside the CAN controller hardware.
See chapter 7.4 for closer description of L-PDU transmission.

On L-PDU reception, the CAN driver calls the RX indication callback function with ID, DLC and pointer to L-SDU as parameter.
See chapter 7.5 for closer description of L-PDU reception.

The CAN driver provides an interface that serves as periodical processing function, and must be called by the CAN interface periodically.

Furthermore the CAN driver 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 driver 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 driver modules implement the interrupt service routines for all CAN Hardware Unit interrupts that are needed. All unused interrupts in the CAN controller shall be disabled by the CAN driver. The CAN driver is responsible to reset the interrupt flag at the end of the ISR (if not done automatically by hardware). The configuration (i.e. priority) and the vector table entry is not done by the CAN driver.

CAN079: All design and implementation guidelines as described in [11] shall be fulfilled for a CAN driver implementation.

7.1 Driver State Machine

The CAN driver has a very simple state machine which is shown in Figure 7.1.

CAN103: After reset the driver is in the state CAN_UNINIT until the function Can_Init is called.

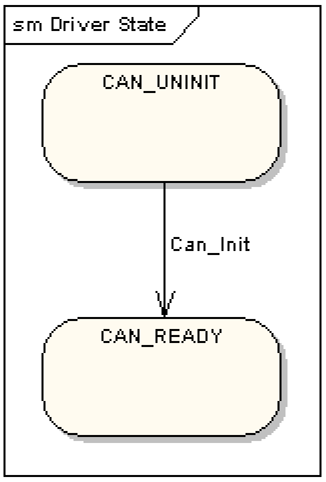


Figure 7-1

After calling Can_Init all CAN controllers are initialized according their configuration.
All Controllers are in the state CAN_CS_STOPPED (see description in chapter 7.2).
Hardware register settings that have impact on all CAN controllers inside the HW Unit can only be set in the function Can_Init.
Each CAN controller must be started separately by calling the function Can_SetControllerMode(CAN_T_START).
After all controllers inside the HW Unit have been initialized the driver state changes to CAN_READY.
Can_Init shall only be called once during runtime. A call of Can_Init in driver state CAN_READY shall be ignored.

The driver must only implement a variable for the driver state, when the development error tracing is switched on. When the development error tracing is switched off, the CAN driver 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 driver function (CAN_E_UNINIT development error).

7.2 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 CAN interface. [5] shows the implemented software state machine. Any CAN hardware access is encapsulated by CAN driver functions, but the CAN driver does not memorize the state changes.

➔ During a transition phase the software controller state inside the CAN interface may differ from the hardware state of the CAN controller.

The CAN driver 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 driver 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 CAN interface with the corresponding callback function. The software state is then changed inside this callback function.

➔ The CAN driver does not check for validity of state changes.
It is the task of CAN interface to trigger only transitions that are allowed in the current state. Only for development errors the transition has to be checked and in case of wrong implementation of CAN interface the development error CAN_E_TRANSITION is raised to the Development Error Tracer.

➔ The CAN driver 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 CAN interface 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 driver does not provide a notification for this case.

7.2.1 State Description

This chapter describs the required hardware behavior for the different SW states. The software state machine itself is implemented and described in the CAN interface. 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, that means it participates in the network. For many controllers leaving the 'initialization'-mode causes the controller to be started.

CANIF_CS_SLEEP
CAN052: The hardware settings only differ for CAN hardware that support a sleep mode (wake-up over CAN bus directly supported by CAN hardware).
In this case the CAN hardware must be set to a state from which the hardware can be woken over CAN Bus.

For all other chips the hardware state is the same as for CANIF_CS_STOPPED.

7.2.2 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)
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.

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).
The CAN driver has to ensure that any settings which will cause the CAN controller to participate in the network are not set in this state.

Can_SetControllerMode(CAN_T_START)
CANIF_CS_STOPPED -> CANIF_CS_STARTED
software triggered
The hardware registers are set in a way which make the CAN controller participating on the network. The code that performs this transition is non-blocking. Can_SetControllerMode does not wait until the CAN controller is fully operational.

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

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
Sets the bits inside the CAN hardware which make the CAN controller stop participating on the network.
The code that performs the transition is non-blocking. Can_SetControllerMode does not wait until the CAN controller is really switched off.
Still pending messages are cancelled. A cancellation notification is not raised. Hint: Even if the messages are cancelled, there are hardware restrictions and racing problems. So it cannot be guaranteed if the cancelled messages are still processed in this situation or not.

Can_SetControllerMode(CAN_T_SLEEP)
CANIF_CS_STOPPED -> CANIF_CS_SLEEP
software triggered
Put the controller into sleep mode.
The code that performs the transition is blocking.. It returns only when it is assured that the CAN hardware is wakeable.

CAN048: In case of a CAN bus wake-up the function returns with CAN_WAKEUP.
A desired timeout time is to be configured by the user.

Can_SetControllerMode(CAN_T_WAKEUP)
CANIF_CS_SLEEP -> CANIF_CS_STOPPED
software triggered
If sleep mode is not supported, the function has no effect. The controller is already in stopped state. The code that performs the transition is non-blocking.

Hardware Wakeup (triggered by wake-up event from CAN bus)
CANIF_CS_SLEEP -> CANIF_CS_STOPPED
triggered by incoming L-PDUs
The CAN interface is notified with the callback function CanIf_ControllerWakeup
This state transition will only occur when sleep mode is supported by hardware.
The CAN driver must validate the wake-up. Validation criterion is that a valid L-PDU has been received.
The code that is executed for that transition is either in interrupt context or in the context of Can_MainFunction_Wakeup.
The L-PDU that caused the wake-up is not further processed.

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 CAN interface is notified with the callback function CanIf_ControllerBusOff after stopped state is reached.

After bus-off detection, the driver must ensure that the CAN controller doesn't participate on the network anymore. Automatic bus-off recovery must be disabled or suppressed. Still pending messages are cancelled. A cancellation notification is not raised.

7.3 CAN Driver/Controller Initialization

The CAN driver needs to be initialized with the function Can_Init before it can be used. The function Can_Init initializes:

- static variables, including flags,
- Common setting for the complete CAN HW unit
- CAN controller specific settings for each CAN controller

CAN056: Post-Build configuration elements that are marked as 'multiple' in chapter 10 can be selected by passing the pointer 'Config' to the init function of the module.

CAN054: Registers that contain ‘overall’ settings also relevant for other driver modules (i.e. SPAL) are initialized in a way that other modules are not affected (BSW12058). Write access to these registers must be performed in an atomic manner.

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

CAN023: The consistency of the configuration must be checked by the configuration tool(s).

CAN053: Registers of CAN controller Hardware resources that are not used are not changed by Can_Init.

Each CAN controller can be re-initialized with the function Can_InitController.
The CAN interface must first set the CAN controller in CANIF_CS_STOPPED state. Then it may call Can_InitController.
Only register areas that contain specific configuration for a single CAN controller are affected by this function.

CAN021: The desired CAN controller configuration can be selected with the parameter Config. 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 driver configuration defines the global CAN HW Unit settings and references to the default CAN controller configuration sets.

The CAN interface must call Can_Init during startup phase. Controller re-initialization may be performed (with Can_InitController) any time as long as the controller is in state CANIF_CS_STOPPED.

7.4 L-PDU transmission

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

CAN059: The L-SDU is not modified. The highest byte in the L-SDU buffer is sent first.

CAN100: Several TX hardware objects with unique HTHs may be configured. The CAN interface provides the HTH as parameter of the TX request. See Figure 7-2 for a possible configuration.

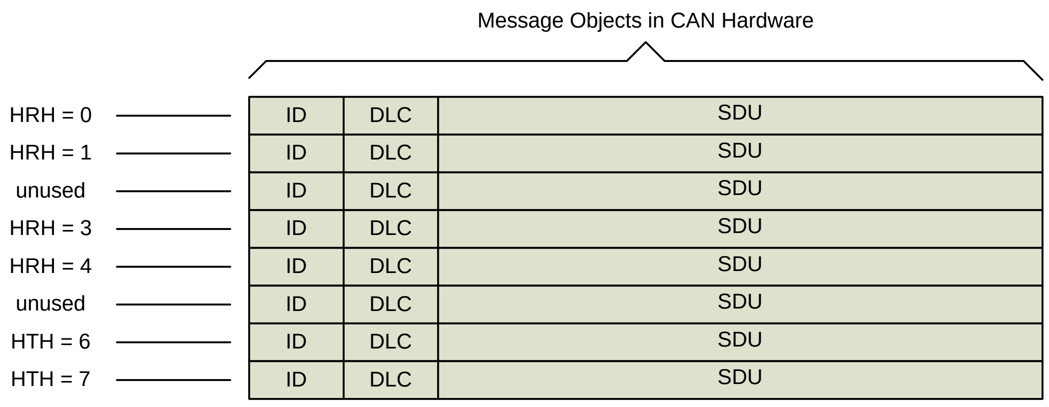


Figure 7-2: Example of assignment of HTHs and HRHs to the Hardware Objects. The numbering of HTHs and HRHs are implementation specific. The chosen numbering is only an example.

The CAN driver writes ID and DLC in the appropriate hardware registers and triggers the transmission.

The CAN driver stores the swPduHandle that is given inside the parameter PduInfo until it calls the CanIf_TXConfirmation for this request where the swPduHandle is given as parameter. This feature is used to reduce time for searching in the CAN Interface implementation.

7.4.1 Priority Inversion

CAN114: To prevent priority inversion two mechanisms are necessary multiplexed transmit and hardware cancellation (see chapter 2.1). These functionalities shall be statically configurable (ON | OFF) at pre-compile time.

7.4.1.1 Multiplexed Transmission

CAN101: Multiplexed transmission mechanisms shall only be supported for devices where either
- multiple transmit buffers can be filled over the same register set, and the µC stores the L-PDU into a free buffer autonomously
- HW supported registers or functions to identify a free buffer are provided.
- L-PDUs of the multiplexed Transmit Objects are send in order of L-PDU priority.

CAN076: Software emulation for multiplexed transmission shall not be implemented.

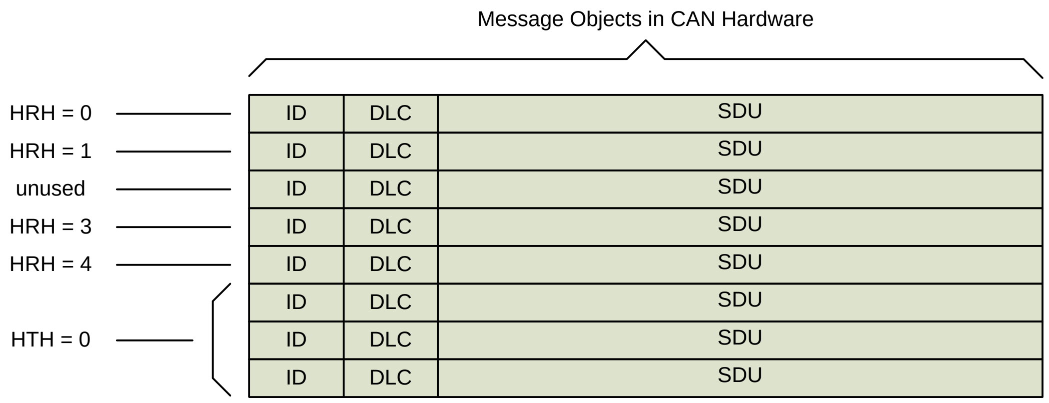


Figure 7-3: Example of assignment of HTHs and HRHs to the Hardware Objects with multiplexed transmission. The numbering of HTHs and HRHs are implementation specific. The chosen numbering is only an example.

7.4.1.2 Transmit Cancellation

CAN097: Transmit cancellation may only be used when no queue per PDU is configured for the CAN interface. It shall only be supported for controllers that support transmit cancellation and multiplexed transmission.

The complete cancellation sequence is described in the CAN interface. The CAN driver initiates a cancellation when all available hardware transmit objects are busy and an L-PDU with higher priority shall be transmitted. The incoming request is also rejected because the cancellation is asynchronous. The CAN driver raises a notification when the cancellation was successful by calling the function CanIf_CancelTxConfirmation. The TX request for the new L-PDU will be repeated by the CAN interface, inside this notification. Immediately after return of the callback function, the CAN driver releases the transmit buffer for new transmissions.

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.

7.4.2 Data Consistency

CAN011: The CAN driver directly uses the buffer of the upper layer.

It is the responsibility of the upper layer to keep the buffer consistent.

The function CanIf_Transmit copies the L-SDU either directly in the CAN Hardware or buffers it if CAN Hardware transmit resources are presently not available.
➔ The L-SDU source buffer (provided by source layer, e.g. COM) can be written with the next value as soon as CanIf_Transmit returns.
➔ The source layer (i.e. COM) is responsible that the L-SDU buffer is not overwritten until the function CanIf_Transmit returned.
➔ SDU must be protected by the layer that calls CanIf_Transmit

7.5 L-PDU reception

On L-PDU reception, the CAN driver calls the RX indication callback function with ID, DLC and pointer to the L-SDU buffer as parameter.

If necessary the ID and DLC are converted to a standardized format (i.e. MSB that marks extended identifiers).

CAN060: The L-SDU is not modified. The byte that is received first is the lowest byte in the L-SDU buffer.

If the RX Buffer of the CAN Hardware is not globally accessible, or if the RX Buffer is not locked by hardware after reception, the CAN driver copies the L-SDU in a shadow buffer.

CAN012: Data consistency issues:
1) The complete RX processing (including copying to destination layer, e.g. COM) is done in context of the RX interrupt or in context of the Can_MainFunction_Read.
➔ When the callback function CanIf_RxIndication returns the L-SDU has already been copied from the CAN hardware (or shadow) buffer to the destination buffer by the destination layer.
➔ As long as it is guaranteed that neither the ISRs nor 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.

2) 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.
The configuration check must assure that the interrupt latency or Can_MainFunction_Read call period can't exceed the time for the reception of one L-PDU.

7.6 Notification concept

CAN057: The CAN driver offers only an event triggered notification interface to the CAN interface. 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 must be possible to implement the driver such that no interrupts at all are used (complete polling).

The configuration of what is and is not polled by the CAN driver is internal to the driver, and not visible outside the module. The polling is done inside the scheduled 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 scheduled 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 scheduled functions Can_MainFunction_Read, Can_MainFunction_Write, Can_MainFunction_BusOff and Can_MainFunction_Wakeup.

7.7 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.

CAN038: Transmit requests are simply forwarded by the CAN interface 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 CAN interface queues that request. (same behavior as if all hardware objects are busy).

CAN090: 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 scheduled main functions (i.e. Can_MainFunction Read) shall not be interrupted by themselves. This must be ensured by the OS. Therefore these scheduled main functions are is not reentrant.

7.8 Error classification

CAN104: The CAN driver 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_CONFIG CAN_E_PARAM_HANDLE CAN_E_PARAM_PDUINFO 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

7.8.1 Development Errors

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

CAN028: Development errors cause the call of the Development Error Tracer when DET is switched on.

CAN091: After return of the DET the function that raised the development error shall return immediately.

CAN089: Development errors are only indicated in return values 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.

7.8.2 Production Errors

CAN029: Hardware errors and failures cause a call of a central error function of the Diagnostic Event Manager.

The Syntax is Dem_SetEventStatus(EventId, EventStatus)

The only error that is reported to DEM by the CAN driver is CAN_E_TIMEOUT. Depending on the CAN hardware, a change of setting may be taken over only after a delay. In that case the CAN driver must i.e. poll a bit until the change has been made in hardware and then return. This polling may take only a limited number of polls (configurable).

When this time is elapsed the error CAN_E_TIMOUT is raised.

This affects the complete COM stack.

In case of a CAN_E_TIMEOUT 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 that raised the production error shall return immediately.

CAN093: The function that raised the production error shall return with CAN_NOT_OK.

7.8.3 Return Values

CAN_BUSY is reported via return value of the function Can_Write. The CAN interface reacts according the sequence diagrams specified for the CAN interface.

CAN_WAKEUP 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.

7.9 Error detection

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

CAN083: If the CAN_DEV_ERROR_DETECT switch is enabled API parameter checking is enabled. The detailed description of the detected errors can be found in chapter 7.8 and chapter 8.

CAN084: The detection of production code errors cannot be switched off.

7.10 Error notification

CAN027: Detected development errors will be reported to the error hook of the Development Error Tracer (DET) if the pre-processor switch CAN_DEV_ERROR_DETECT is set (see chapter 10). No code for catching development errors shall be generated, when development errors are switched off.

7.11 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 drivers as described in **CANIF124** in [5].

8.1 Imported types

8.1.1 Standard types

The following list shows all types of Std_Types.h that are used by the CAN driver

- uint8

8.1.2 ComStack types

The following list shows all types of ComStack_Types.h that are used by the CAN driver

- PduIdType

8.2 Type definitions

8.2.1 Can_ConfigType

Type:	structure
Range:	-- --
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

Type:	structure
Range:	-- --
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.

8.2.3 Can_PduType

Type:	structure
Range:	-- --

Description:	This type is used to provide ID, DLC and SDU from CAN interface to CAN driver
	{
	Can_IdType id;

	uint8 length;
	uint8 *sdu;
	PduIdType swPduHandle;
	}

8.2.4 Can_IdType

Type:	configuration dependent uint16: if only Standard IDs are used uint32: if also Extended IDs are used	
Range:	0...0x7FF	for Standard IDs
	0...0xFFFFFFFF	for Extended IDs
Description:	Represents the Identifier of an L-PDU. For extended IDs the most significant bit is set.	

8.2.5 Can_StateTransitionType

Type:	enumeration	
Range:	CAN_T_START	see 7.2.2
	CAN_T_STOP	see 7.2.2
	CAN_T_SLEEP	see 7.2.2
	CAN_T_WAKEUP	see 7.2.2
Description:	State transitions that are used by the function CAN_SetControllerMode	

8.2.6 Can_ReturnType

CAN039:

Type:	enumeration	
Range:	CAN_OK	success
	CAN_NOT_OK	error occured
	CAN_BUSY	transmit request could not be processed because no transmit object was available
	CAN_WAKEUP	wakeup event occurred during sleep transition
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

Service name:	Can_Init

Syntax:	`void Can_Init`
	`(`
	`uint8 CtrlIdx, const Can_ConfigType *Config`
	`)`

Service ID [hex]:	0x00	
Sync/Async:	Synchronous	
Reentrancy:	non reentrant	
Parameters (in):	`Config`	Pointer to driver configuration
	`CtrlIdx`	index for multiple (selectable) configuration parameter sets
Parameters (out):	none	
Return value:	none	
Description:	**CAN008:** Driver Module Initialization. Initializes: ▪ static variables, including flags ▪ CAN HW Unit global hardware settings ▪ CAN controller specific settings All CAN controllers are in state CANIF_CS_STOPPED after initialization. (see also state machine descriptions in chapters 7.1 and 7.2) This function may raise the production error CAN_E_TIMEOUT if the configuration changes are not overtaken by hardware after a configured number of polls. Development errors: CAN_E_TRANSITION: the driver is not in 'uninitialized' state CAN_E_PARAM_CONFIG: returned in case a NULL pointer was given as config parameter. Production errors: CAN_E_TIMEOUT – indicates that initialization could not be performed (indicates defective hardware)	
Caveats:	This service shall be called before any other service of a CAN driver	
Configuration:	Symbolic names of the available configuration sets are provided by the configuration description of the CAN driver. See chapter 10 about configuration description.	

8.3.1.2 Can_GetVersionInfo

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 (out):	`versioninfo`	Pointer to where to store the version information of this module.
Return value:	None	
Description:	**CAN105:** This service returns the version information of this module. The version information includes: - Module Id - Vendor Id - Vendor specific version numbers (BSW00407).	

	Hint: If source code for caller and callee of this function is available this function should be realized as a macro. The macro should be defined in the modules header file.
Caveats:	None
Configuration:	This function is pre compile time configurable `On/Off` by the configuration parameter: `CAN_VERSION_INFO_API`

8.3.2 Services affecting one single CAN Controller

8.3.2.1 Can_InitController

Service name:	Can_InitController	

Syntax:	`void Can_InitController`
	`(`
	`uint8 Controller, 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 (out):	none	
Return value:	none	
Description:	**CAN062:** CAN controller (re-)initialization. Different sets of static configuration may have been configured. The parameter *Config points to the hardware specific structure that describes the configuration. This function initializes only CAN controller specific settings. 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. The CAN controllers is in state CANIF_CS_STOPPED after (re-)initialization. (see also state machine description in chapter 7.2) Development errors: CAN_E_UNINIT: Driver not yet initialized CAN_E_PARAM_CONFIG: `Config` is a null pointer CAN_E_PARAM_CONTROLLER: Parameter `Controller` is out of range CAN_E_TRANSITION: the controller is not 'stopped' Production errors: CAN_E_TIMEOUT – indicates that initialization could not be performed (indicates defective hardware)	
Caveats:	none	
Configuration:	Symbolic names of the available configuration sets are provided by the configuration description of the CAN driver. See chapter 10 about configuration description.	

8.3.2.2 Can_SetControllerMode

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`	for possible transitions see section 7.2.2
Return value:	`CAN_OK`	transition initiated
	`CAN_NOT_OK`	development or production error occured
	`CAN_WAKEUP`	a wakeup during transition to 'sleep' occured
Description:	**CAN017:** Performs software triggered state transitions of the CAN controller State machine. See also [BSW12169] Each possible transition is related to a specified sequence of CAN controller register modifications. This function returns always with CAN_OK. Exeption: Only for CAN_T_SLEEP transition the function may return CAN_WAKEUP, when wakeup event from CAN bus occurred during sleep transition. Precondition: During the function executes the wake-up interrupt must be disabled, so that the wake-up status can be checked inside this function. For all other state changes, the function does not wait until the state change has really been performed. Anyway this function is asynchronous because the actual result may occur later. But neither callback nor notification will report the actual state change afterwards. This function enables interrupts that are needed in the new state. It disables interrupts that are not allowed in the new state. Note: enabling and disabling may not be executed, when CAN interrupts are disabled with CAN_DisableControllerInterrupts. Development errors: CAN_E_UNINIT: Driver not yet initialized CAN_E_PARAM_CONTROLLER: Parameter `Controller` is out of range CAN_E_TRANSITION: invalid transition has been requested. The CAN interface is responsible not to initiate invalid transitions. Production errors: CAN_E_TIMEOUT – indicates that initialization could not be performed (indicates defective hardware, not for sleep transition)	
Caveats:	The behavior of Transmit operation is undefined while 'software' state in CAN interface is already CANIF_CS_STARTED, but CAN controller is not yet in operational mode. The CAN interface must ensure that the function is not called for the same controller before the previous call of Can_SetControllerMode returned.	
Configuration:	none	

8.3.2.3 Can_DisableControllerInterrupts

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 (out):	none	
Return value:	none	
Description:	**CAN049:** This function disables all interrupts for this CAN controller. When Can_DisableControllerInterrupts is called several times (without calling Can_EnableControllerInterrupts in between) only the first has any effect on hardware. Further calls of Can_DisableControllerInterrupts increase a counter that indicates how many Can_ControllerEnableInterrupts need to be called before the interrupts will be enabled (incremental disable) Individual enabling and disabling of interrupts in other functions (i.e. Can_SetControllerMode) shall be tracked, 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. Development Errors: CAN_E_UNINIT: Driver not yet initialized CAN_E_PARAM_CONTROLLER: Parameter `Controller` is out of range	
Caveats:	--	
Configuration:	None	

8.3.2.4 Can_EnableControllerInterrupts

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 (out):	none	

Return value:	none	
Description:	**CAN050:** This function enables all interrupts that shall be enabled according the current software status. When Can_DisableControllerInterrupts has been called several times, Can_EnableControllerInterrupts must be called as many times before the interrupts are re-enabled. No action shall be performed when Can_DisableControllerInterrupts has not been called before. See also implementation example for Can_DisableControllerInterrupts. Development Errors: CAN_E_UNINIT: Driver not yet initialized CAN_E_PARAM_CONTROLLER: Parameter `Controller` is out of range	
Caveats:	--	
Configuration:	none	

8.3.3 Services affecting a Hardware Handle

8.3.3.1 Can_Write

Service name:	Can_Write	

Syntax:	`Can_ReturnType Can_Write`
	`(`
	`uint8 Hth, Can_PduType *PduInfo`
	`)`	

Service ID [hex]:	0x06	
Sync/Async:	Synchronous	
Reentrancy:	reentrant (thread-safe)	
	`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 (out):	none	
Return value:	`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:	**CAN015:** 1) Can_Write checks if hardware transmit object that is identified by the HTH is free. Can_Write checks if another Can_Write is ongoing for the same HTH, this synchronization is implemented with mutexes that are either provided by OS or implemented by the CAN driver. a) 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 b) hardware transmit object is busy with another transmit request ▪ The function returns with CAN_BUSY c) A preemtive call of Can_Write has been issued, that could not be handled reentrant (i.e. a call with the same HTH). ▪ The function returns with CAN_BUSY -> the function is non blocking d) the hardware transmit object is busy with another transmit request for an L-PDU that has lower priority than that for the current request ▪ The transmission of the previous L-PDU is cancelled (asynchronusly). ▪ The function returns with CAN_BUSY Development Errors: CAN_E_UNINIT: Driver not yet initialized CAN_E_PARAM_HANDLE: Parameter `Hth` is not a configured Hardware Transmit Handle CAN_E_PARAM_PDUINFO: Parameter `PduInfo` is a null-pointer Production Errors: none
Caveats:	none
Configuration:	The numbers of the available HTHs are described in the configuration description file.

8.4 Call-back notifications

The CAN driver 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

Service name:	Can_MainFunction_Write
Syntax:	`void Can_MainFunction_Write ( void )`
Service ID [hex]:	0x01
Description:	**CAN031:** This function performs the polling of TX confirmation and TX cancellation confirmation that are configured statically as 'to be polled'.

	Can_MainFunction_Write might be implemented as empty define in case no polling at all is used. Development Errors: CAN_E_UNINIT: Driver not yet initialized An OS development error may be raised, when the task that calls the Can_MainFunction_Write has been activated more than once.
Timing:	variable cyclic
Pre condition:	--
Configuration:	The implementation of this functions strongly depends on the static polling configuration.

8.5.1.2 Can_MainFunction_Read

Service name:	Can_MainFunction_Read
Syntax:	`void Can_MainFunction_Read ( void )`
Service ID [hex]:	0x08
Description:	**CAN108:** This function performs the polling of RX indications that are configured statically as 'to be polled'. Can_MainFunction_Read might be implemented as empty define in case no polling at all is used. Development Errors: CAN_E_UNINIT: Driver not yet initialized An OS development error may be raised, when the task that calls the Can_MainFunction_Read has been activated more than once.
Timing:	variable cyclic
Pre condition:	--
Configuration:	The implementation of this functions strongly depends on the static polling configuration.

8.5.1.3 Can_MainFunction_BusOff

Service name:	Can_MainFunction_BusOff
Syntax:	`void Can_MainFunction_BusOff ( void )`
Service ID [hex]:	0x09
Description:	**CAN109:** This function performs the polling of bus-off events that are configured statically as 'to be polled'. Can_MainFunction_BusOff might be implemented as empty define in case no polling at all is used. Development Errors: CAN_E_UNINIT: Driver not yet initialized An OS development error may be raised, when the task that calls the Can_MainFunction_BusOff has been activated more than once.
Timing:	variable cyclic
Pre condition:	--
Configuration:	The implementation of this functions strongly depends on the static polling

	configuration.

8.5.1.4 Can_MainFunction_Wakeup

Service name:	Can_MainFunction_Wakeup
Syntax:	`void Can_MainFunction_Wakeup ( void )`
Service ID [hex]:	0x10
Description:	**CAN112:** This function performs the polling of wake-up events that are configured statically as 'to be polled'. Can_MainFunction_Wakeup might be implemented as empty define in case no polling at all is used. Development Errors: CAN_E_UNINIT: Driver not yet initialized An OS development error may be raised, when the task that calls the Can_MainFunction_Wakeup has been activated more than once.
Timing:	variable cyclic
Pre condition:	--
Configuration:	The implementation of this functions strongly depends on the static polling configuration.

Terms and definitions:

Fixed cyclic: Fixed cyclic means that one cycle time is defined at configuration and shall not be changed because functionality is requiring that fixed timing (e.g. filters).
Variable cyclic: Variable cyclic means that the cycle times are defined at configuration, but might be mode dependent and therefore vary during runtime.
On pre condition: On pre condition means that no cycle time can be defined. The function will be called when conditions are fulfilled. Alternatively, the function may be called cyclically however the cycle time will be assigned dynamically during runtime by other modules.

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.

CAN102: All callback functions that are called by the CAN driver are implemented in the CAN interface. These callback functions are not configurable.

API function	***Module***	***Description***
CanIf_RxIndication	CAN interface	**CAN013:** Indicates that a new L-PDU arrived. Is either called by the RX-interrupt service routine of

API function	***Module***	***Description***
		the corresponding HW resource or inside Can_MainFunction_Read in case of polling mode.
CanIf_TxConfirmation	CAN interface	**CAN016:** Indicates a successful transmission. Is either called by the TX-interrupt service routine of the corresponding HW resource or inside the Can_MainFunction_Write in case of polling mode.
CanIf_CancelTxConfirmation	CAN interface	**CAN098:** Indicates a cancelled transmission. Is called at the end of Can_Write if a pending L-PDU has been cancelled. The implementation of that function is inside the CAN interface. The transmit buffer is released for the next transmission after return of CanIf_CancelTxConfirmation.
CanIf_ControllerWakeup	CAN interface	**CAN018:** ndicates that a wake-up was detected. Is called by the wake-up interrupt service routine of the corresponding controller or by Can_MainFunction_Wakeup, if the wakeup event is polled. The wake-up must be validated by the CAN driver. Wake-up notification shall only be raised, when a valid CAN L-PDU has been received. Restrictions: - Only called if supported by hardware - Prerequisite for external CAN controllers: Wire between Wake-up pin of CAN controller and an appropriate microcontroller port pin. Polling context: Detection of wake-up is typically done with reading a Microcontroller Portpin Interrupt context: Wake-up event raises an interrupt in Microcontroller
CanIf_ControllerBusOff	CAN interface	**CAN019:** Indicates that the controller went in bus-off mode. Is called by the bus-off interrupt service routine of the corresponding controller or by Can_MainFunction_BusOff, if the bus-off event is polled.
Dem_ReportError	Dem	Routine to report production relevant error events by event ID.

8.6.2 Optional Interfaces

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

API function	***Module***	***Description***	***Configuration parameter (description see chapter 10)***
Det_ReportError	Det	Development error notification	CAN_DEV_ERROR_DETECT

8.6.3 Configurable interfaces

There is no configurable target for the CAN driver. CAN driver always reports to CAN interface.

9 Sequence diagrams

For Sequence diagrams see the CAN interface Specification [5].
There the complete sequences for Transmission, Reception and Error Handling are described.

10 Configuration specification

This chapter defines configuration parameters and their clustering into containers. In order to support the specification Chapter 10.1 describes fundamentals.

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

Chapter 10.3 specifies published information of the module CAN driver.

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 described parameters are input for the CAN driver configurator.

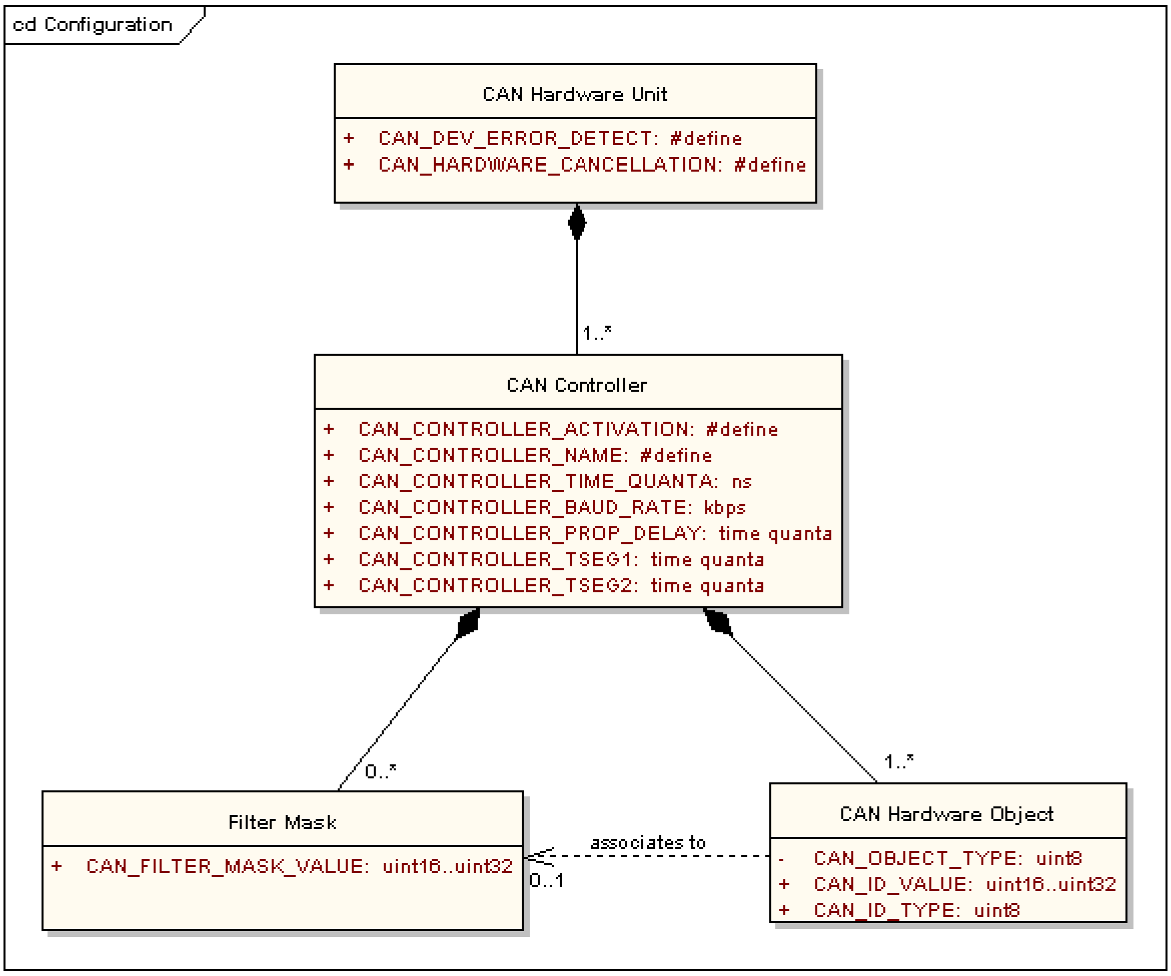
CAN022: The code configurator of the CAN driver is CAN controller specific.
If the CAN controller is sited on-chip, the code generation tool for the CAN driver is µController specific.
If the CAN controller is an external device the generation tool must not be µController specific.

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.).

The CAN driver provides two variants of configuration sets:
VariantPC: all variables are pre-compile time configurable

VariantPB: Mix of precompile and Post Build multiple selectable configurable configurations.


cd Configuration
CAN Hardware Unit
+ CAN_DEV_ERROR_DETECT: #define
+ CAN_HARDWARE_CANCELLATION: #define
1..*
CAN Controller
+ CAN_CONTROLLER_ACTIVATION: #define
+ CAN_CONTROLLER_NAME: #define
+ CAN_CONTROLLER_TIME_QUANTA: ns
+ CAN_CONTROLLER_BAUD_RATE: kbps
+ CAN_CONTROLLER_PROP_DELAY: time quanta
+ CAN_CONTROLLER_TSEG1: time quanta
+ CAN_CONTROLLER_TSEG2: time quanta
0..*
1..*
Filter Mask
+ CAN_FILTER_MASK_VALUE: uint16..uint32
associates to
0..1
CAN Hardware Object
- CAN_OBJECT_TYPE: uint8
+ CAN_ID_VALUE: uint16..uint32
+ CAN_ID_TYPE: uint8

10.2.1.1 CanDriverConfiguration

SWS Item	
Container Name	CanDriverConfiguration
Description	This container contains the configuration (parameters) of the CAN driver.
Configuration Parameters	

Name	CAN_DEV_ERROR_DETECT		
Description	**CAN064:** Switches the Development Error Detection and Notification ON or OFF.		
Type	#define		
Unit	--		
Range	ON	enabled	
	OFF	disabled	
Configuration Class	***Pre-compile***	x	all Variants
	Link time	--	--
	Post Build	--	--
Scope	Module		
Dependency	None		

Name	`CAN_VERSION_INFO_API`		
Description	**CAN106:** Switches the Can_GetVersionInfo function `ON` or `OFF`.		
Type	`#define`		
Unit	--		
Range	`ON`	enabled	
	`OFF`	disabled	
Configuration Class	***Pre-compile***	x	all Variants
	Link time	--	--
	Post Build	--	--
Scope	module		
Dependency	none		

Name	`CAN_MULTIPLEXED_TRANSMISSION`		
Description	**CAN095:** : Specifies if multiplexed transmission shall be supported. ON or OFF		
Type	`#define`		
Unit	--		
Range	`ON`	enabled	
	`OFF`	disabled	
Configuration Class	***Pre-compile***	x	all Variants
	Link time	--	--
	Post Build	--	--
Scope	module		
Dependency	CAN Device supports multiplexed transmission		

Name	`CAN_HARDWARE_CANCELLATION`		
Description	**CAN069:** Specifies if hardware cancellation shall be supported. ON or OFF		
Type	`#define`		
Unit	--		
Range	`ON`	enabled	
	`OFF`	disabled	
Configuration Class	***Pre-compile***	x	all Variants
	Link time	--	--
	Post Build	--	--
Scope	module		
Dependency	CAN interface is configured to support hardware cancellation.		

Name	`CAN_TIMEOUT_DURATION`		
Description	**CAN113:** Specifies the maximum number of loops for blocking function until a timeout is raised in short term wait loops.		
Type	`#define`		
Unit	--		
Range	`system specific`	--	
Configuration Class	***Pre-compile***	x	all Variants
	Link time	--	--
	Post Build	--	--
Scope	module		
Dependency	--		

Included Containers

Container Name	***Multiplicity***	***Scope***	***Dependency***
CanController	1..*	ECU	--

10.2.1.2 CanController

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

Name	`CAN_CONTROLLER_ACTIVATION`		
Description	Defines if a CAN controller is used in the configuration.		
Type	`#define`		
Unit	--		
Range	`ON`	CAN controller is used	
	`OFF`	CAN controller is not used	
Configuration Class	***Pre-compile***	x	all Variants
	Link time	--	--
	Post Build	--	--
Scope	ECU		
Dependency	None		

Name	`CAN_CONTROLLER_NAME`		
Description	Identifies the controller (within an ECU) by a symbolic name. This name shall be used by the CAN driver users and must therefore be published.		
Type	`#define`		
Unit	--		
Range	`ANSI C compliant name`	--	
Configuration Class	***Pre-compile***	x	all Variants
	Link time	--	--
	Post Build	--	--
Scope	ECU		
Dependency	None		

Name	`CAN_CONTROLLER_TIME_QUANTA`		
Description	**CAN063:** 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		
Type	`float`		
Unit	`ns`		
Range	`hardware specific`	--	
Configuration Class	***Pre-compile***	x	VariantPC
	Link time	--	--
	Post Build	M	VariantPB
Scope	Network		
Dependency	None		

Name	`CAN_CONTROLLER_BAUD_RATE`		
Description	**CAN005:** Specify the baud rate of the controller		
Type	`uint16`		
Unit	`kbps`		
Range	`125, 500, 1000`	--	
Configuration Class	***Pre-compile***	x	VarriantPC
	Link time	--	--
	Post Build	M	VariantPB
Scope	Network		
Dependency	None		

Name	`CAN_CONTROLLER_PROP_DELAY`		
Description	**CAN073:** Propagation delay		
Type	`uint8`		
Unit	`time quantas`		
Range	`hardware specific`	--	
Configuration Class	***Pre-compile***	x	VariantPC
	Link time	--	--
	Post Build	M	VariantPB
Scope	Network		
Dependency	None		

Name	`CAN_CONTROLLER_TSEG1`		
Description	**CAN074:** Specifies TSEG1		
Type	`uint8`		
Unit	`time quantas`		
Range	`hardware specific`	--	
Configuration Class	***Pre-compile***	x	Variant PC
	Link time	--	--
	Post Build	M	VariantPB
Scope	Network		
Dependency	None		

Name	`CAN_CONTROLLER_TSEG2`		
Description	**CAN075:** Specifies TSEG2		
Type	`uint8`		
Unit	`time quantas`		
Range	`hardware specific`	--	
Configuration Class	***Pre-compile***	x	VariantPC
	Link time	--	--
	Post Build	M	VariantPB
Scope	Network		
Dependency	None		

Included Containers			
Container Name	***Multiplicity***	***Scope***	***Dependency***
CanFilterMask	0..`number of HW Objects`	ECU	--
CanHardwareObject	1..*	ECU	--

Note: An additional configuration is the polling mode. But the configuration possibilitiies are strongly MCU dependent and therefore this configuration will not be part of the standard configuration.

10.2.1.3 CanFilterMask

SWS Item	
Container Name	CanFilterMask
Description	This container contains the configuration (parameters) of the CAN Filter Mask(s).
Configuration Parameters	
Name	`CAN_FILTER_MASK_VALUE`
Description	**CAN066:** Describes a mask for hardware-based filtering of CAN

	identifiers It shall be distinguished between - Standard identifier mask - Extended identifier mask.		
Type	`unit16..uint32`		
Unit	--		
Range	`11 Bit`	standard id mask	
	`29 Bit`	extended id mask	
Configuration Class	***Pre-compile***	x	VariantPC
	Link time	--	--
	Post Build	M	VariantPB
Scope	ECU		
Dependency	The filter mask settings must be known by the CAN interface configurator for optimization of SW the filters		

Included Containers			
Container Name	***Multiplicity***	***Scope***	***Dependency***
none	--	--	--

10.2.1.4 CanHardwareObject

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

Name	`CAN_OBJECT_TYPE`		
Description	Specifies if the HardwareObject is used as Transmit or as Receive object		
Type	`uint8`		
Unit	--		
Range	RX	Object is used for reception	
	TX	Object is used for transmission	
Configuration Class	***Pre-compile***	x	VariantPC
	Link time	--	--
	Post Build	M	VariantPB
Scope	ECU		
Dependency	None		

Name	`CAN_OBJECT_HANDLE_ID`		
Description	Unique number that identifies a hardware object		
Type	`uint8`		
Unit	`--`		
Range	`0..FFh`	--	
Configuration Class	***Pre-compile***	x	VariantPC
	Link time	--	--
	Post Build	--	--
Scope	ECU		
Dependency	None		

Name	`CAN_OBJECT_HANDLE_NAME`
Description	Unique name that identifies a hardware object. This name can be used

	by the CAN Interface to address a hardware object		
Type	`#define`		
Unit	`--`		
Range	`ANSI C compliant name`	--	
Configuration Class	**Pre-compile**	x	VariantPC
	Link time	--	--
	Post Build	M	VariantPB
Scope	ECU		
Dependency	None		

Name	`CAN_MASK_REFERENCE`		
Description	Reference to the filter mask that is used for hardware filtering togerther with the CAN_ID_VALUE		
Type	reference to CanFilterMask		
Unit	`--`		
Range	`--`	--	
Configuration Class	**Pre-compile**	x	VariantPC
	Link time	--	--
	Post Build	M	VariantPB
Scope	ECU		
Dependency	Only relevant when `CAN_OBJECT_TYPE` is RX		

Name	`CAN_ID_VALUE`		
Description	**CAN067:** Specifies (together with the filter mask) the identifiers that pass the hardware filter for of RX objects.		
Type	`uint16..uint32`		
Unit	`--`		
Range	`0h...7FFh`	for 11 bit identifier	
	`0h...1FFFFFFFh`	for 29 bit identifier	
Configuration Class	**Pre-compile**	X	VariantPC
	Link time	--	--
	Post Build	M	VariantPB
Scope	ECU		
Dependency	None		

Name	`CAN_ID_TYPE`		
Description	**CAN065:** Specifies whether the IdValue is of type - standard identifier - extended identifier - mixed mode		
Type	`uint8`		
Unit	`--`		
Range	`Standard`	standard identifier	
	`Extended`	extended identifier	
	`Mixed`	standard and extended identifier	
Configuration Class	**Pre-compile**	x	VariantPC
	Link time	--	--
	Post Build	x	VariantPB
Scope	ECU		
Dependency	None		

Included Containers			
Container Name	**Multiplicity**	**Scope**	**Dependency**
none	--	--	--

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.

SWS Item	**CAN085**	
Information elements		
Information element name	***Type / Range***	***Information element description***
`CAN_VENDOR_ID`	`uint16 / --`	Vendor ID of the dedicated implementation of this module according to the AUTOSAR vendor list
`CAN_MODULE_ID`	`uint8 / --`	Module ID of this module from Module List
`CAN_AR_MAJOR_VERSION`	`uint8 / --`	Major version number of AUTOSAR specification where the appropriate implementation is based on.
`CAN_AR_MINOR_VERSION`	`uint8 / --`	Minor version number of AUTOSAR specification where the appropriate implementation is based on.
`CAN_AR_PATCH_VERSION`	`uint8 / --`	Patch level version number of AUTOSAR specification where the appropriate implementation is based on.
`CAN_SW_MAJOR_VERSION`	`uint8 / --`	Major version number of the vendor specific implementation of the module. The numbering is vendor specific.
`CAN_SW_MINOR_VERSION`	`uint8 / --`	Minor version number of the vendor specific implementation of the module. The numbering is vendor specific.
`CAN_SW_PATCH_VERSION`	`uint8 / --`	Patch level version number of the vendor specific implementation of the module. The numbering is vendor specific.

11 Changes to Release 1

11.1 Deleted SWS Items

SWS Item	***Rationale***
CAN070	Requirement from Volvo/Ford has been restricted to special hardware that support multiplexed transmission
CAN025	replaced by requirements CAN026...CAN029

11.2 Replaced SWS Items

SWS Item of Release 1	***replaced by SWS Item***	***Rationale***
CAN006	CAN076	Requirement from Volvo/Ford has been restricted to special hardware that support multiplexed transmission
CAN005	CAN005 CAN073 CAN074 CAN075	Changed configuration description according new template.
CAN030 CAN040 CAN041 CAN042 CAN045	CAN079	No room for design rules according template -> all design rules packed in one requirement that references the Programming Guidelines

11.3 Changed SWS Items

SWS Item	***Rationale***
CAN036	Bugfix regarding callback functions.
CAN064	Changed configuration description according new template.
CAN069	Changed configuration description according new template.
CAN066	Changed configuration description according new template.
CAN067	Bugfix regarding Full-CAN TX: No differentiation for Basic-CAN TX and Full-CAN TX -> no statically configured ID.
CAN065	Changed configuration description according new template.
CAN031	Separate Main Functions for RX, TX and BusOff (BSW00433)

11.4 Added SWS Items

SWS Item	***Rationale***
CAN077	Required to fulfill BSW00347
CAN078	Part of SWS Template
CAN080	Part of SWS Template
CAN081	Part of SWS Template
CAN082	Part of SWS Template
CAN083	Part of SWS Template
CAN084	Part of SWS Template
CAN085	Part of SWS Template

CAN089	Clarified open issues regarding development error detection and return value
CAN090	Reentrancy of Can_EnableControllerInterrupt and Can_DisableControllerInerrupt
CAN091	Clarified open issues regarding development error detection and function abortion
CAN092	Clarified open issue regarding production error detection and function abortion
CAN093	Clarified open issues regarding production error detection and return value
CAN094	support for external CAN devices
CAN095	configurability of feature multiplexed transmission
CAN097	new requirement for support of TX cancellation
CAN098	new requirement for support of TX cancellation
CAN099	individual configuration of event polling mode per event (feature is not new, but had not an ID.
CAN100	configuration of multiple TX objects
CAN101	multiplexed transmission
CAN102	New WP1.1.2 BSW General Requirement BSW00387
CAN103	No new functionality. Added ID to map on BSW00406
CAN104	No new functionality. Added ID to map on BSW00385
CAN105	New Function to fulfill new Wp1.1.2 requirement BSW00407
CAN106	New configuration parameter to fulfill new Wp1.1.2 requirement BSW00407
CAN108	New Function to fulfill new WP1.1.2 requirement BSW00433
CAN109	New Function to fulfill new WP1.1.2 requirement BSW00433
CAN110	New Function to fulfill new WP1.1.2 requirement BSW00428
CAN111	Version Check
CAN112	New Function to fulfill new WP1.1.2 requirement BSW00433
CAN113	Timeout configuration for short term wait loops
CAN114	Added tag to already existing sentence