VECTOR

MICROSAR Classic CAN Driver

Technical Reference

Version 4.01.00

Contents

Document Information

History

Version	Date	Author	Remarks
3.0.0	28-12-2023	hmalkawi, visgaz	Migrate to rst (CANCORE-2373)
	12-01-2024	visbir	Remove Extended RAM Check feature (CANCORE-2403)
main-1	2024-02-26	visgaz	Change history is maintained in the global ChangeHistory.txt file starting with this release (CANCORE-2371)

Table 1 History

Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	Specification of CAN Driver	R4.0.3
[2]	AUTOSAR	Specification of Development Error Tracer	R4.0.3
[3]	AUTOSAR	List of Basic Software Modules	V1.0.0
[4]	Vector	TechnicalReference_vCan_<Vendor API Infix>.pdf	see delivery
[5]	Vector	MICROSAR Classic Post-Build Loadable	see delivery
[6]	Vector	Technical Reference CAN Interface	see delivery

Caution

We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector's release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

Scope of the Document

This technical reference describes the general use of the CAN Driver basis software. All aspects which are CAN Hardware Unit specific are described in a separate document (see *[4]*), which is also part of the delivery.

1 Introduction

This document describes the functionality, integration, API and configuration of the module Can.

The Can is specified in AUTOSAR *[1]*.

Since each CAN Hardware Unit has its own behavior based on the CAN specifications, the main goal of the CAN Driver is to provide a standardized interface to support communication over the CAN bus. The CAN Driver works closely together with the higher layer CAN Interface.

1.1 Architecture Overview

These are the interfaces of the module:

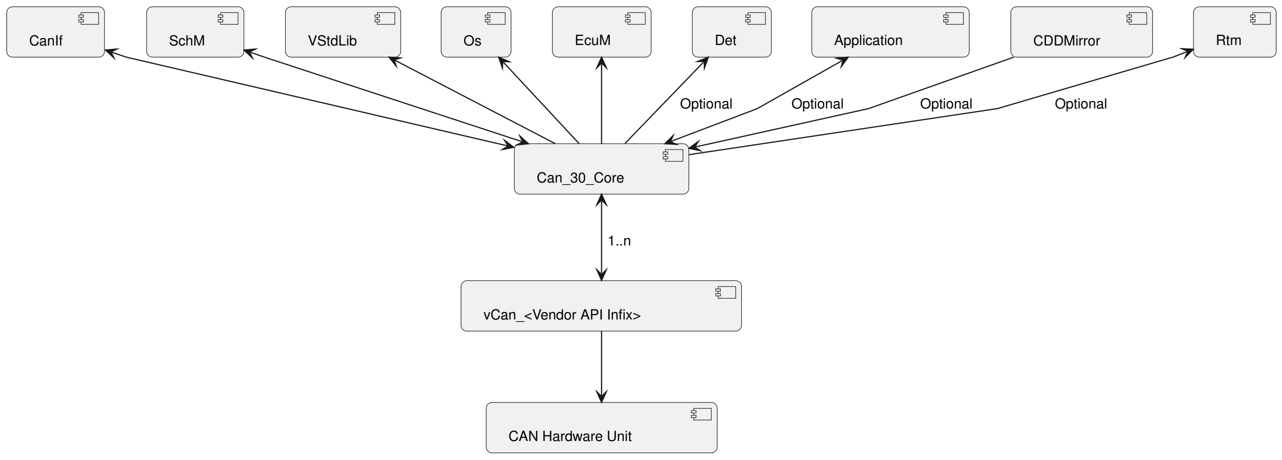


Figure 1.1 Module Architectural Environment

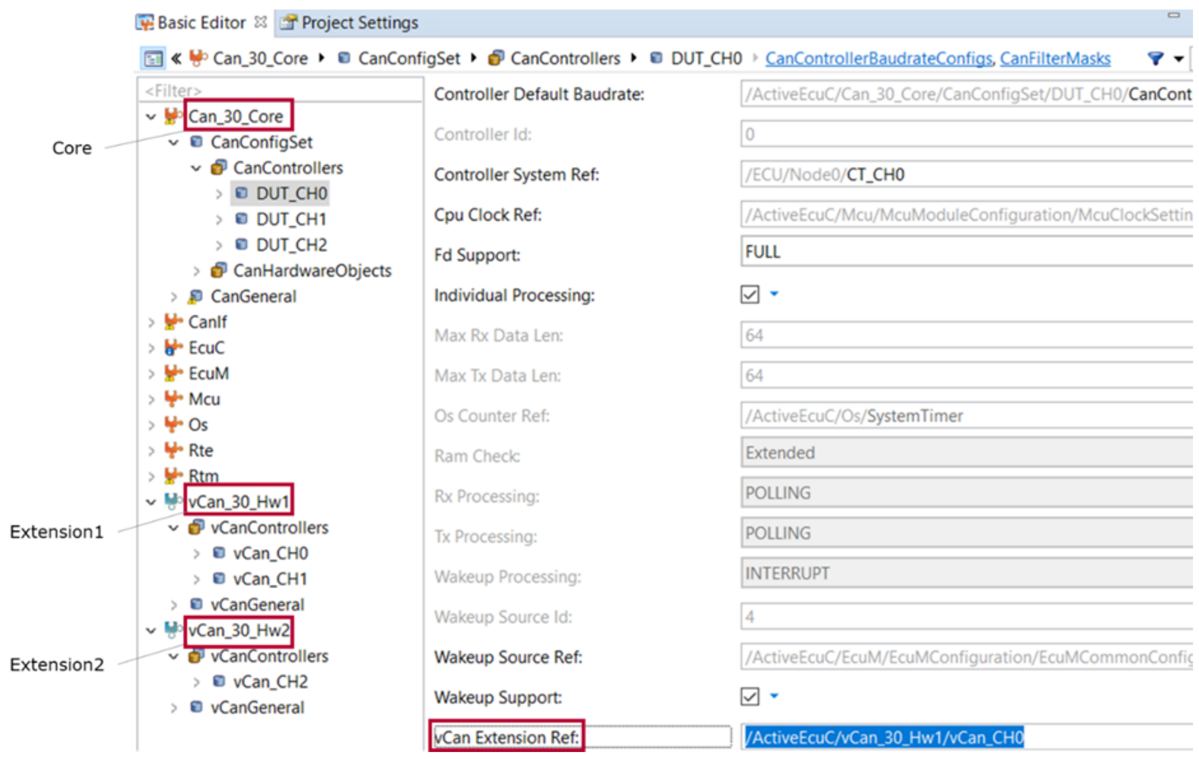
The CAN Driver Core (Can_30_Core) is extended by one or multiple hardware specific modules (`vCan_30_<Vendor Api Infix>`). This allows the Core to manage different hardware specific CAN Hardware Units in parallel.

Multiple vCan modules are used when different CAN Hardware Units are available on one microcontroller.

Each Core and vCan module have their own separate implementation, generated code, and configuration namespaces including BSWMDs.

All hardware independent setting as defined by AUTOSAR and enriched by Vector, can be found in the Core configuration namespace. Hardware specific configuration settings can be found in the vCan module namespace.

The vCan modules are referred by the Core on `CanController`-container level as a link, so that each `CanController` of the Core can be mapped to a dedicated `vCanController` of an vCan module (see figure below, where 'Hw1' and 'Hw2' represent different CAN hardware units).


Core
Extension1
Extension2
Basic Editor
Project Settings
Can_30_Core ▸ CanConfigSet ▸ CanControllers ▸ DUT_CH0 ▸ CanControllerBaudrateConfigs, CanFilterMasks
<Filter>

- Can_30_Core
  - CanConfigSet
    - CanControllers
      - DUT_CH0
      - DUT_CH1
      - DUT_CH2
    - CanHardwareObjects
  - CanGeneral
- CanIf
- EcuC
- EcuM
- Mcu
- Os
- Rte
- Rtm
- vCan_30_Hw1
  - vCanControllers
    - vCan_CH0
    - vCan_CH1
  - vCanGeneral
- vCan_30_Hw2
  - vCanControllers
    - vCan_CH2
  - vCanGeneral

Controller Default Baudrate:	/ActiveEcuC/Can_30_Core/CanConfigSet/DUT_CH0/CanCont
Controller Id:	0
Controller System Ref:	/ECU/Node0/CT_CH0
Cpu Clock Ref:	/ActiveEcuC/Mcu/McuModuleConfiguration/McuClockSettin
Fd Support:	FULL
Individual Processing:	✓
Max Rx Data Len:	64
Max Tx Data Len:	64
Os Counter Ref:	/ActiveEcuC/Os/SystemTimer
Ram Check:	Extended
Rx Processing:	POLLING
Tx Processing:	POLLING
Wakeup Processing:	INTERRUPT
Wakeup Source Id:	4
Wakeup Source Ref:	/ActiveEcuC/EcuM/EcuMConfiguration/EcuMCommonConfig
Wakeup Support:	✓
vCan Extension Ref:	/ActiveEcuC/vCan_30_Hw1/vCan_CH0

Figure 1.2 Core/vCan configuration

For detailed description of the configuration parameters please refer to the BSWMD file of the component (Core or vCan) which can be seen in the configuration tool’s GUI as well.

2 Functional Description

2.1 Features

The features listed in the following sections cover the functionality specified for the CAN Driver.

The AUTOSAR standard functionality is specified in *[1]*, the corresponding features are listed below.

2.1.1 Supported AUTOSAR Standard Conform Features

- CAN Driver initialization

Communication

- Transmitting PDUs in corresponding CAN frames
- Transmit confirmation - Callback function for successful transmission
- Transmit cancellation (`CanHardwareCancellation`)
  - Cancellation of PDUs (out of CAN RAM) to avoid internal priority inversion.
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- Receiving CAN frames
- Receive indication - Callback function for received CAN frames as PDUs
- Overrun notification (`CanOverrunNotification`)
  - A detected overrun or overwrite at a Rx `CanHardwareObject` of type Basic- or Full-CAN is notified to the DET module (regarding to *[1]*) or to the application (via `Appl_30_Core_CanOverrun` (for Basic-CANs) or `Appl_30_Core_CanFullCanOverrun` (for Full-CANs)).
  - Whether a CanHardwareObject has overrun or overwrite behavior and whether a overrun or overwrite can be detected depends on the underlying CAN Hardware Unit(s) (see *[4]*).

CAN controller modes

- Sleep mode (power saving)
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- Stop mode (passive to CAN bus)
- Start mode (active to CAN bus)
- Bus off event
- Wake-up event (over CAN bus) (`CanWakeupSupport, CanWakeupSourceRef`)
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).

Polling modes

- Polling mode for transmit confirmation (`CanTxProcessing`)
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- Polling mode for reception (`CanRxProcessing`)
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- Polling mode for wake-up event (`CanWakeupProcessing`)
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- Polling mode for bus off event (`CanBusoffProcessing`)
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- Polling mode for mode transition

CanHardwareObjects

- Tx Basic-CAN
  - `CanHardwareObject` to send PDUs as CAN frames with different CAN IDs (can be assigned to a Tx-Buffer from CAN Interface).
- Tx Basic-CAN of type Multiplexed TX
  - `CanHardwareObject` that groups up to three different PDUs to avoid external priority inversion when transmitting.
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- Tx Full-CAN
  - `CanHardwareObject` to send PDUs as CAN frame with one specific CAN-ID.
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- Rx Full-CAN - `CanHardwareObject` to receive CAN frames with one specific CAN-ID. - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- Rx Basic-CAN
  - `CanHardwareObject` to receive CAN frames with different CAN IDs.
  - Realization in CAN RAM (FIFO or shadow buffer) depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- Standard CAN identifier
- Extended CAN identifier
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- Standard and extended CAN identifier (Mixed CAN identifier types)
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).

Others

- Error detection and notification over DET module (`CanDevErrorDetection`)
- API to read version information of this module (`CanVersionInfoApi`)
- Hardware Loop Check (`CanHardwareLoopCheck`)
  - Timeout monitoring to avoid possible endless loops (e.g., due to a hardware issue) executed by CAN Driver or by application.
  - Need and realization depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- Multiple CAN Drivers
  - Multiple CAN Drivers are supported through infixing of the Driver's APIs.

2.1.2 Features Provided Beyond the AUTOSAR Standard

- Initialization of memory at power on.
- Mirror mode (`CanMirrorModeSupport`)
  - Pass transmitted and received PDUs to the CDDMirror module. Mirroring can be enabled/disabled CAN controller specific.
- Security Event Reporting (CanEnableSecurityEventReporting)
  - Provides CAN controller state and error detection to support security event reporting according to AR 20-11.
- Silent mode (`CanSilentModeSupport`)
  - A CAN controller in this mode only passively listens to the CAN bus.
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- CAN FD Mode1 (`CanFdSupport / BRS`)
  - Support of CAN FD frames with baudrate switch based on AR 4.2.0.
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- CAN FD Mode2 (`CanFdSupport / FULL`)
  - Support of CAN FD frames with baudrate switch and up to 64 data bytes based on AR 4.2.0.
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- Individual Polling (`CanIndividualProcessing`)
  - Polling mode can be enabled CanHardwareObject specific.
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
  - HighEnd License is required.

- Multiple Rx Basic-CANs
  - Provides the possibility to use multiple Rx Basic-CANs with multiple filters to optimize acceptance filtering and to avoiding overrun(s).
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
  - HighEnd License is required.
- Rx Queue (`CanRxQueue / CanRxQueueSize`)
  - The Rx Queue provides the possibility to store received CAN frames as PDUs in interrupt context and handle them asynchronous in polling task. This reduces the interrupt service execution time.
  - HighEnd License is required.
- Multiple Tx Basic-CANs
  - Multiple Tx Basic-CANs can be used to send different groups of PDUs over separate Basic-CANs with different software buffering behavior (for software buffering CAN Interface see *[6]*).
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
  - HighEnd License is required.
- Tx Basic-CAN of type HwFIFO (`CanObjectHwFifo`)
  - CanHardwareObject using a FIFO mechanism with configurable size to send PDUs.
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
  - HighEnd License is required.
- Nested CAN interrupts
  - Activation/deactivation of higher priority CAN interrupts that may interrupt an already active CAN interrupt.
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- Mixed CAN identifier types (`CanSupportExtendedID`, `CanSupportMixedID`)
  - Force the CAN Driver to be able to handle standard and extended CAN IDs at Pre-Compile time to be able to add PDUs with extended CAN IDs at Post-Build time.
- Generic Pre-Copy (`CanGenericPrecopy`)
  - Callout function that is called for each received PDU (following CAN Interface callback can be suppressed).
- Generic Confirmation (`CanGenericConfirmation`, `CanGenericConfirmationAPI2`)
  - Callout function that is called for successful transmitted PDU (following CAN Interface callback can be suppressed).

- Generic Pre-Transmit (`CanGenericPreTransmit`)
  - Callout function that is called for each PDU to be transmitted in order to modify the PDU before transmission.
- Get Hardware Status
  - Provides APIs to get hardware status information (see `Can_30_Core_GetStatus()`, `Can_30_Core_GetControllerErrorState()`, `Can_30_Core_GetControllerMode()`, `Can_30_Core_GetControllerRxErrorCounter()`, `Can_30_Core_GetControllerTxErrorCounter()`).
  - Support of the different status information depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- Interrupt Category selection
  - Provides support for category 2 Interrupt Service Routines in OS.
- RAM Check
  - Provides a check of the CAN RAM for each CanHardwareObject.
  - Support depends on the underlying CAN Hardware Unit(s) (see *[4]*).
- Multi Partition
  - CAN controllers can be mapped to different partitions.
  - Support and limitations depends on the underlying CAN Hardware Unit(s) (see *[4]*).
  - Multi Partition License is required.
- Baud rate
  - CAN controller baud rate can be configured with a floating point (according to AR 4.4.0) or integer (according to *[1]*) number.

2.2 Communication

Note
All CAN Hardware Unit dependent information about communication, e.g., the layout in the CAN RAM for the `CanHardwareObjects` is provided in *[4]*.

2.2.1 Transmission

`Can_30_Core_Write()` is used to send a PDU over a `CanHardwareObject` in a CAN frame. The data, DLC and CAN ID are copied into related CAN RAM area and a send request is set. In addition before the send request is set the CAN ID, DLC and data of the passed PDU can be changed via the `Appl_30_Core_GenericPreTransmit()` callout function (if enabled).

On each sent request the CAN ID will always be copied into related CAN RAM independent whether the corresponding `CanHardwareObject` is a Basic- or Full-CAN.

If the requested `CanHardwareObject` is currently occupied, the status busy will be returned. Then the PDU can be queued in a Tx Buffer of the CAN Interface (if feature is enabled - see *[6]*).

If cancellation in hardware is supported and enabled the PDU with lowest priority CAN ID inside a `CanHardwareObject` is cancelled and re-queued in the CAN Interface.

A successful transmission is by default confirmed to the CAN Interface by calling `CanIf_TxConfirmation()` callback function. If the feature Generic Confirmation is enabled, the `Appl_30_Core_GenericConfirmation()` callout function will be called at first. The following call of `CanIf_TxConfirmation()` can be suppressed by `Appl_30_Core_GenericConfirmation()` return value.

2.2.2 Reception

A received CAN frame is per default indicated as a PDU to the CAN Interface by calling the `CanIf_RxIndication()` callback function. If the feature Generic Pre-Copy is enabled the `Appl_30_Core_GenericPreCopy()` callout function will be called at first. The following call of `CanIf_RxIndication()` can be suppressed by `Appl_30_Core_GenericPreCopy` () return value.

If the feature Rx Queue is enabled, the received CAN frames (polling or interrupt context) will be queued as PDUs (same queue over all CAN controllers). The Rx Queue will be read by calling `Can_30_Core_Mainfunction_Read()` and the Rx Indication (like `CanIf_RxIndication()`) will be called out of this context. Rx Queue is used for interrupt based systems to keep the interrupt latency time short.

2.3 CAN controller modes

The mode of a CAN controller can be changed via `Can_30_Core_SetControllerMode()`. The last requested transition will be executed. The upper layer should take care about valid transitions.

Note
A detailed overview of which mode transitions are supported by a CAN controller is described in *[4]*.

Caution
Sleep mode emulation: If wake-up support is deactivated by configuration, the CAN controller goes into `Stop` mode instead of `Sleep` for a requested transition to `Sleep` mode.

2.4 Multi-Partition

The CAN controllers (`Can/CanConfigSet/CanController`) can be mapped to different partitions via the parameter `CanIf/CanIfCtrlDrvCfg/CanIfCtrlCfg/CanIfEcucPartitionRef` in the CAN Interface (see *[6]*).

Depending on the underlying CAN Hardware Unit (refer to *[4]*) there may be the restriction that not all CAN controllers can be mapped individually or independently.

Each partition can be further mapped to a (physical) Core. This offers the possibility of load balancing because the processing for the CAN controllers can then be executed in parallel.

The initialization of the CAN Driver must always be executed on the main Core.

Refer to *Main Functions* to view changes in main function API when Multi-partition is in use.

The following features are not supported when combined with Multi-partition:

- Rx Queue
- Runtime Measurement
- Security Event Reporting
- Mirror mode

2.5 Error Handling

2.5.1 Development Error Reporting

By default, development errors are reported to the DET using the service `Det_ReportError()` as specified in *[2]*, if development error reporting is enabled.

The reported service IDs and error IDs are contained and documented in the header file Can_30_Core_Types.h.

2.5.2 Hardware Loop Check / Timeout Monitoring

The feature "Hardware Loop Check" is used to break endless loops caused by software or hardware issues. The synchronous part of mode transitions will be also handled by this timeout mechanism, which is no issue but only a timing limit (The potential following asynchronous part of the transition is handled without the "Hardware Loop Check"). Refer to the *Uncritical Loops* subsection.

If "Hardware Loop Check by Application" is not activated the check is handled internally by the CAN Driver. Nevertheless, refer to "Short Description" in the table below as there may be activities that should be initiated by the application (e.g., reset of CAN controller or special mode transitions). If actions are necessary, the feature "Hardware Loop Check by Application" can be used to handle the corresponding loops like described.

2.5.2.1 Critical Loops

The CAN Driver Core does not have any critical loops. Please refer to *[4]* for the CAN Driver vCan module specific critical loops.

2.5.2.2 Uncritical Loops

No additional application handling needed after the loop breaks.

Loop name/Source	Short description
CAN_30_CORE_LOOP_MODE_CHANGE	This CAN controller dependent loop is called in `Can_30_Core_SetControllerMode()` and is processed if the CAN Hardware Unit does not enter the corresponding CAN controller mode. Refer to chapter "Controller Modes" in *[4]* to learn more about potential reasons for delay. The loop is used for a short transition blocking (short synchronous timeout) without differentiation for different mode transitions. There is no issue or additional handling necessary if the loop is canceled as the CAN Driver handles the mode transitions asynchronously after this synchronous timeout is reached.

Table 2.1 Uncritical Loops

2.5.2.3 CAN RAM Check

The CAN Driver supports a check of the CAN RAM for all used `CanHardwareObjects` (ID, DLC, data). The CAN RAM check is called internally every time a power on is executed within initialization, or a CAN bus wake-up event happens. The CAN Driver verifies that no used `CanHardwareObject` is corrupt. A `CanHardwareObject` is considered corrupt if a predefined pattern is written to the appropriate CAN RAM and the read operation does not return the expected pattern. If a corrupt `CanHardwareObject` is found the callout function `Appl_30_Core_CanCorruptMailbox()` is called. This function tells the application which `CanHardwareObject` is corrupt.

After the check of all `CanHardwareObjects` the CAN Driver calls the callout function `Appl_30_Core_CanRamCheckFailed()` if at least one corrupt `CanHardwareObject` was found. The application must decide if the CAN Driver disables communication or not by means of the callout function's return value. If the application has decided to disable the communication, there is no possibility to enable the communication again until the next call to `Can_30_Core_Init()` or a wake-up event occurs.

2.6 Compliance

2.6.1 Unsupported Features

These are the major unsupported features compared to AUTOSAR *[1]*

Functional

> CAN Hardware Unit as external device
> Inter Module Checks

Config

> `CanSupportTTCANRef`

2.6.2 Deviations

These are the major deviations compared to AUTOSAR *[1]*

Functional

- `CanHardwareCancellation` / `CanIdenticalIdCancellation`
  - Only PDUs with the same PDU ID are canceled.
  - Deviation only relevant if a feature for dynamic CAN IDs from the CAN Interface is used otherwise CAN Driver behaves as specified by *[1]*.
  - Support at all depends on the underlying CAN Hardware Unit(s) (see *[4]*).

API

- <LPDU_CalloutName>
  - Realized via similar feature `CanGenericPrecopy` (configuration parameter `CanGeneral`/ `CanGenericPrecopy`).
- `Can_CheckWakeup()`
  - Return type `Can_ReturnType` or `Std_ReturnType` is used depending on configuration parameter `CanCheckWakeupCanRetType`.

Config

- `CanMainFunctionReadPeriod` / `CanMainFunctionWritePeriod`
  - Only one write and read period is supported.
- `CanMainFunctionRWPeriodRef`
  - Realized via similar feature `CanIndividualProcessing` (configuration parameter `CanHardwareObject`/ `CanHwProcessing`).

3 Integration

This chapter gives necessary information for the integration of the MICROSAR Classic CAN Driver into an application environment of an ECU.

3.1 Embedded Implementation

The delivery of the module contains these source code files, some files are generated by the configuration tool DaVinci Configurator Classic:

File Name	Description	Integration Tasks
Can_30_Core.c	Source file of the module.	-
Can_30_Core.h	Header file that defines services used by typically upper layer modules.	-
Can_30_Core_vCan.h	Header file that defines callbacks used by typically lower layer modules.	-
Can_30_Core_<unit>.c	Source file of the unit	-
Can_30_Core_<unit>.h	Header file that defines services used by other units.	-
Can_30_Core_Cfg.h	Generated header file that contains pre-compile switches which configure the module features. It also declares pre-compile configurable data.	-
Can_30_Core_PBcfg.c	Generated source file that defines post-build changeable data.	-
Can_30_Core_Lcfg.c	Generated source file that defines pre-compile configurable data.	-
Can_30_Core_Types.h	Header file that defines types definitions, constants, and macros used in the module's API.	-
Can_30_Core_ PrivateTypes.h	Header file that defines types definitions, constants, and macros used by the module's internals.	-
Can_ DrvGeneralTypes.h	Generated header file that defines types definitions, constants, and macros used by the module's internals.	If the generated types need to be replaced, for example when using a 3rd party driver, the integrator can remove the dependence on this file in `Can_GeneralTypes.h` and replace it with their own definitions.

Table 3.1 Source Code Files

3.2 Initialization

`Can_30_Core_Init()` has to be called to initialize the CAN Driver at power on and initialize all CAN controllers.

After this call all CAN controllers will stay in `Stop` mode until the CAN Interface changes it to `Start`.

`Can_30_Core_InitMemory()` is an additional service function to reinitialize the memory to bring the CAN Driver back to a pre-power-on state (not initialized). Afterwards `Can_30_Core_Init()` has to be called again. It is recommended to use this function before calling `Can_30_Core_Init()` to secure that no startup-code specific pre-initialized variables affect the CAN Driver startup behavior.

Caution
The used vCan modules must be initialized via `vCan_30_<Vendor Api Infix>_Init()` before the Core initialization via `Can_30_Core_Init()` is performed.

3.3 Main Functions

The APIs `Can_30_Core_MainFunction_Write()`, `Can_30_Core_MainFunction_Read()`, `Can_30_Core_MainFunction_BusOff()`, and `Can_30_Core_MainFunction_Wakeup()` are called by the upper layer to poll events if the specific polling mode is activated. Otherwise these functions return without action and the events will be handled in interrupt context.

When individual polling is activated only CanHardwareObjects that are configured for individual polling will be polled in the main functions `Can_30_Core_MainFunction_Write()` and `Can_30_Core_MainFunction_Read()`. Whereas all others are handled in interrupt context.

When the Rx Queue is activated, the queue is filled in interrupt or polling context depending on the configuration. But the processing (indications) will be done in `Can_30_Core_MainFunction_Read()` context.

`Can_30_Core_MainFunction_Mode()` is called by the upper layer to poll asynchronous mode transition notifications.

If feature Multi-partition is activated, the main function names are extended by the OsApplication name (like: `Can_30_Core_MainFunction_<Read/Write/BusOff/Wakeup/Mode>_<OsApplicationName>()`).

3.4 Synchronization

3.4.1 Critical Sections

To protect internal data structures against unwanted modification, the module uses critical sections for blocking concurrent access.

The AUTOSAR standard provides with the BSW Scheduler a BSW module, which handles entering and leaving critical sections.

For more information about the BSW Scheduler please refer to *[3]*. When the BSW Scheduler is used the CAN Driver provides critical section codes that have to be mapped by the BSW Scheduler to following mechanism:

Critical Section Define	Description
`CAN_30_CORE_EXCLUSIVE_AREA_1`	Used inside `Can_30_Core_DisableControllerInterrupts()` and `Can_30_Core_EnableControllerInterrupts()` to avoid nested calls of these APIs. This ensures consistency while modifying the interrupt counter and CAN interrupt lock registers. > Duration is SHORT; the interrupt registers and a counter variable are modified. > No API call of other BSW inside. > Disable global interrupts *or* Empty in case the functions are called within a context with a higher or equal priority to the CAN interrupt, and the calls are not nested (within each other or themselves).
`CAN_30_CORE_EXCLUSIVE_AREA_2`	Used inside `Can_30_Core_Write()`, `Can_30_Core_CancelTx()` and `Can_30_Core_Confirmation()` to secure software states of `CanHardwareObjects`. OR When `Appl_30_Core_GenericConfirmation()` is called to prevent the use of `Can_30_Core_Write()` which ensures data consistency. > Duration is MEDIUM, because of: Time needed for Tx confirmation, multiplexed Tx and cancellation handling; cancellation handling triggered by CAN Interface; and the modification of CAN Tx registers, and semaphores of locked `CanHardwareObjects`. > The area contains a call to `CanIf_CancelTxConfirmation()`, which contains no further calls to other CanIf functions. > Disable global interrupts *or* Disable CAN interrupts and do not call function `Can_30_Core_Write()` and `Can_30_Core_CancelTx()` reentrant.
`CAN_30_CORE_EXCLUSIVE_AREA_4`	Used inside received data handling (Rx Queue treatment) to secure Rx Queue counter and data. > Duration is SHORT; Rx Queue counter will be modified, and data will be copied to Rx Queue. > No API call of other BSW inside. > Disable Global Interrupts *or* Disable all CAN interrupts.

continues on next page

Table 3.2 – continued from previous page

Critical Section Define	Description
`CAN_30_CORE_EXCLUSIVE_AREA_5`	Used inside wake-up handling to secure state transition. (Only in wake-up polling mode) Wake-up polling must not be interrupted by any mode change like triggered by `Can_30_Core_SetControllerMode()`, bus off handling, `Can_30_Core_Init()`, `Can_30_Core_ChangeBaudrate()` or `Can_30_Core_SetBaudrate()`. > Duration is MEDIUM; wake-up mode transition will be performed. > Call to DET inside. > Use CAN interrupt locks here, in case the above mentioned APIs are only called within same task level or CAN interrupt context. *or* No action (locks) necessary when all APIs (mentioned above) are called from same context or cannot interrupt each other. *or* Disable global interrupts
`CAN_30_CORE_EXCLUSIVE_AREA_6`	Used inside `Can_30_Core_SetControllerMode()` and bus off to avoid nested state transition requests. `Can_30_Core_SetControllerMode()` and bus off handling (polling or interrupt) must not be interrupted by other calls of `Can_30_Core_SetControllerMode()`, bus off handling, `Can_30_Core_Init()`, `Can_30_Core_ChangeBaudrate()` or `Can_30_Core_SetBaudrate()`. > Duration is MEDIUM; mode transition will be performed. > No API call of other BSW inside. > Use CAN interrupt locks here, in case the above mentioned APIs are only called within same task level or CAN interrupt context. *or* No action (locks) necessary when all APIs (mentioned above) are called from same context or cannot interrupt each other. *or* Disable global interrupts (please refer to 'Note' below)
`CAN_30_CORE_EXCLUSIVE_AREA_7`	Used inside transmit data handling (Tx HwFIFO treatment) to secure Tx HwFIFO counter and data. > Duration is SHORT; Tx HwFIFO counter will be modifed, and data will be copied to Tx HwFIFO . > No API call of other BSW inside. > Disable Global Interrupts *or* Disable all CAN interrupts.

Table 3.2 Critical Section Codes

Note
The CAN driver uses the OS APIs `GetCounterValue()` and `GetElapsedValue()` like defined by AUTOSAR for timeout monitoring.
This timeout monitoring is used within the `CAN_30_CORE_EXCLUSIVE_AREA_6` (mode change) and may also be used within the vCan module and upper layers critical sections as well like `CANIF_EXCLUSIVE_AREA_0`, `CANSM_EXCLUSIVE_AREA_1`, `CANSM_EXCLUSIVE_AREA_4` and `COMM_EXCLUSIVE_AREA_1` (Refer to 'Critical Section' chapter in the related Technical references).
Therefore the usage of 'Global Interrupt Lock' within these AREAs is limited.
Using 'Global Interrupt Lock' combined with these OS APIs called, will lead to the issue `E_OS_DISABLEDINT` notified by `ErrorHook()`.
In order to avoid this issue there are two possibilities:
1.) Do not use 'Global Interrupt Lock' for these critical sections. But use the alternatives like described for these critical sections.
(E.g. for `CAN_30_CORE_EXCLUSIVE_AREA_6` `Can_30_Core_DisableControllerInterrupts()` & `Can_30_Core_EnableControllerInterrupts()` for all used controllers, could be used instead. Please refer to the limitations described in the critical section AREA_6 above.)
2.) Activate feature
`Can_30_Core/Can/CanGeneral/CanHardwareCancelByAppl` and implement the timeout monitoring callouts according to the given API description in the BSWMD for this feature. The monitoring could be implemented by using a free running timer or just a counter loop waiting for the expected repetitions. (see also chapter *Uncritical Loops*)

3.5 Memory Sections

The module does not require any special integrational task for its memory sections.

3.6 Callouts

The module has callout functions, for which an implementation has to be provided based on their *API description*.

3.7 Error Reporting Det

If another module is used for development error reporting, the function prototype for reporting the error can be configured by the integrator, but must have the same signature as the service `Det_ReportError()`.

The module has these properties used for the error reporting to the Det.

> CAN_MODULE_ID = 80
> CAN_VENDOR_ID = 30

The reported service IDs and error IDs are contained and documented in the header file

Can_30_Core_Types.h.

4 API Description

4.1 Provided Functions

The module provides these service functions.

Service functions are implemented by the module and are typically called by upper-layer modules. They are declared in Can_30_Core.h.

4.1.1 Can_30_Core_InitMemory

Prototype
void **Can_30_Core_InitMemory**( )
Parameters
None
Return codes
N/A
Functional Description
Initializes component variables in _*INIT*_ sections at power up. Use this to re-run the system without performing a new start from power on. (E.g.: used to support an ongoing debug session without a complete re-initialization.)
Particularities
Module is not initialized or in STOP mode.
Options
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE

Table 4.1 Can_30_Core_InitMemory

4.1.2 Can_30_Core_Init

Prototype	
void **Can_30_Core_Init**(Can_30_Core_ConfigPtrType ConfigPtr)	
Parameters	
ConfigPtr	Component configuration structure.
Return codes	
N/A	
Functional Description	
Initializes all component variables and sets the component state to initialized.	
Particularities	
> Interrupts are disabled. > Module is uninitialized. > Can_30_Core_InitMemory() has been called unless CAN Module is initialized by start up code.	
Options	

continues on next page

Table 4.2 – continued from previous page

Callcontext: ANY Reentrant: FALSE Synchronous: TRUE

Table 4.2 Can_30_Core_Init

4.1.3 Can_30_Core_GetVersionInfo

Prototype	
void **Can_30_Core_GetVersionInfo**(Can_30_Core_VersionInfoPtrType VersionInfo)	
Parameters	
VersionInfo	Pointer to the version information. Parameter must not be NULL and available for write access.
Return codes	
N/A	
Functional Description	
Returns version information, vendor ID and AUTOSAR module ID.	
Particularities	
Feature has to be configured.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.3 Can_30_Core_GetVersionInfo

4.1.4 Can_30_Core_ChangeBaudrate

Prototype	
Std_ReturnType **Can_30_Core_ChangeBaudrate**(uint8 Controller, uint16 Baudrate)	
Parameters	
Controller	CAN Controller to be changed.
Baudrate	Baud rate to be set.
Return codes	
E_NOT_OK Baud rate was not set. E_OK Baud rate was set.	
Functional Description	
Change baud rate. This service shall change the baud rate and re-initialize the CAN controller.	
Particularities	
> Feature has to be configured. > The CAN controller must be in "STOP" mode.	
Options	

continues on next page

Table 4.4 – continued from previous page

Callcontext: TASK Reentrant: FALSE Synchronous: TRUE

Table 4.4 Can_30_Core_ChangeBaudrate

4.1.5 Can_30_Core_SetBaudrate

Prototype	
Std_ReturnType **Can_30_Core_SetBaudrate**(uint8 Controller, uint16 BaudRateConfigID)	
Parameters	
Controller	Controller to be set.
BaudRateConfigID	Identity of the configured baud rate (available as Symbolic Name).
Return codes	
E_NOT_OK Baud rate was not set. E_OK Baud rate was set.	
Functional Description	
Set baud rate. The service shall change the baud rate and re-initialize the CAN controller.	
Particularities	
> Feature has to be configured. > The CAN controller must be in "STOP" mode.	
Options	
Callcontext: TASK Reentrant: FALSE Synchronous: TRUE	

Table 4.5 Can_30_Core_SetBaudrate

4.1.6 Can_30_Core_CheckBaudrate

Prototype	
Std_ReturnType **Can_30_Core_CheckBaudrate**(uint8 Controller, uint16 Baudrate)	
Parameters	
Controller	CAN controller to be checked.
Baudrate	Baud rate to be checked.
Return codes	
E_NOT_OK Baud rate is not available. E_OK Baud rate is available.	
Functional Description	
Checks the baud rate. This service shall check if a certain CAN controller supports a requested baudrate.	
Particularities	

continues on next page

Table 4.6 – continued from previous page

> Feature has to be configured. > The CAN controller must be initialized.
Options
Callcontext: TASK Reentrant: FALSE Synchronous: TRUE

Table 4.6 Can_30_Core_CheckBaudrate

4.1.7 Can_30_Core_SetControllerMode

Prototype	
Can_ReturnType **Can_30_Core_SetControllerMode**(uint8 Controller, Can_StateTransitionType Transition)	
Parameters	
Controller	The requested CAN controller.
Transition	The requested transition mode (CAN_T_START, CAN_T_STOP, CAN_T_SLEEP, CAN_T_WAKEUP).
Return codes	
CAN_NOT_OK Transition request rejected. CAN_OK Transition request accepted.	
Functional Description	
Requests a mode transition that will be performed synchronously within a short time. The mode change might be finished asynchronously within the Can_30_Core_MainFunction_Mode() if not here.	
Particularities	
Must not be called within CAN driver context (Rx, Tx or Bus off callouts).	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: FALSE	

Table 4.7 Can_30_Core_SetControllerMode

4.1.8 Can_30_Core_GetControllerMode

Prototype	
Std_ReturnType **Can_30_Core_GetControllerMode**(uint8 Controller, Can_30_Core_ControllerStatePtr-Type ControllerModePtr)	
Parameters	
Controller	The requested CAN controller
ControllerModePtr	Pointer to variable in which the driver will store the CAN controller mode. Parameter must not be NULL and available for write access.
Return codes	

continues on next page

Table 4.8 – continued from previous page

E_NOT_OK Controller mode request was not accepted. E_OK Controller mode request was accepted.
Functional Description
Returns the mode of the given CAN controller.
Options
Callcontext: ANY Reentrant: TRUE Synchronous: TRUE

Table 4.8 Can_30_Core_GetControllerMode

4.1.9 Can_30_Core_SetMirrorMode

Prototype	
void **Can_30_Core_SetMirrorMode**(uint8 Controller, CddMirror_MirrorModeType State)	
Parameters	
Controller	The requested CAN controller.
State	The requested state.
Return codes	
No return value (void)	
Functional Description	
Sets the mirror mode state of the requested CAN controller.	
Particularities	
Feature has to be configured.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.9 Can_30_Core_SetMirrorMode

4.1.10 Can_30_Core_GetMirrorMode

Prototype	
CddMirror_MirrorModeType **Can_30_Core_GetMirrorMode**(uint8 Controller)	
Parameters	
Controller	The requested CAN controller.
Return codes	
CDDMIRROR_INACTIVE Mirror mode is inactive. CDDMIRROR_ACTIVE Mirror mode is active.	
Functional Description	
Returns the mirror mode state of the requested CAN controller.	
Particularities	

continues on next page

Table 4.10 – continued from previous page

Feature has to be configured.
Options
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE

Table 4.10 Can_30_Core_GetMirrorMode

4.1.11 Can_30_Core_SetSilentMode

Prototype	
void **Can_30_Core_SetSilentMode**(uint8 Controller, Can_30_Core_SilentModeType SilentMode)	
Parameters	
Controller	The requested CAN controller.
SilentMode	The requested silent mode state.
Return codes	
No return value (void)	
Functional Description	
Sets the silent mode state of the requested CAN controller.	
Particularities	
Feature has to be configured.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.11 Can_30_Core_SetSilentMode

4.1.12 Can_30_Core_GetSilentMode

Prototype	
Can_30_Core_SilentModeType **Can_30_Core_GetSilentMode**(uint8 Controller)	
Parameters	
Controller	The requested CAN controller.
Return codes	
CAN_30_CORE_SILENT_INACTIVE Silent mode is inactive. CAN_30_CORE_SILENT_ACTIVE Silent mode is active.	
Functional Description	
Returns the Silent mode state of the requested CAN controller.	
Particularities	
Feature has to be configured.	
Options	

continues on next page

Table 4.12 – continued from previous page

Callcontext: ANY Reentrant: FALSE Synchronous: TRUE

Table 4.12 Can_30_Core_GetSilentMode

4.1.13 Can_30_Core_Write

Prototype	
Can_ReturnType **Can_30_Core_Write**(Can_HwHandleType hth, Can_PduInfoPtrType PduInfo)	
Parameters	
hth	Handle of the CanHardwareObject intended to transmit the PDU.
PduInfo	Pointer to the PDU information. Caller must ensure the consistency of the PDU information.
Return codes	
CAN_NOT_OK Transmit request rejected. CAN_OK Transmit request successful. CAN_BUSY Transmit request could not be accomplished due to the CanHardwareObject is busy.	
Functional Description	
Requests a transmission of a PDU as CAN frame.	
Particularities	
> CAN interrupts has to be disabled/locked (Refer to critical section). > Controller must be in Start mode.	
Options	
Callcontext: ANY Reentrant: TRUE Synchronous: TRUE	

Table 4.13 Can_30_Core_Write

4.1.14 Can_30_Core_CancelTx

Prototype	
void **Can_30_Core_CancelTx**(Can_HwHandleType Hth, PduIdType PduId)	
Parameters	
Hth	Handle of the CanHardwareObject intended to be cancelled.
PduId	PDU identifier.
Return codes	
N/A	
Functional Description	
Cancel the TX PDU out of CAN RAM (if possible) or mark the PDU as not to be confirmed in case of the cancellation is unsuccessful.	
Particularities	

continues on next page

Table 4.14 – continued from previous page

Feature has to be configured.
Options
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE

Table 4.14 Can_30_Core_CancelTx

4.1.15 Can_30_Core_CheckWakeup

Prototype	
Std_ReturnType/Can_ReturnType **Can_30_Core_CheckWakeup**(uint8 Controller)	
Parameters	
Controller	The requested CAN controller.
Return codes	
E_OK/CAN_OK The given CAN controller caused a WAKEUP before. E_NOT_OK/CAN_NOT_OK The given CAN controller caused no WAKEUP before.	
Functional Description	
Check the occurrence of WAKEUP events for the given CAN controller (used as WAKEUP callback for higher layers).	
Particularities	
Feature has to be configured.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.15 Can_30_Core_CheckWakeup

4.1.16 Can_30_Core_GetStatus

Prototype	
uint8 **Can_30_Core_GetStatus**(uint8 Controller)	
Parameters	
Controller	CAN controller requested for status information.
Return codes	
CAN_30_CORE_STATUS_START START mode. CAN_30_CORE_STATUS_STOP STOP mode. CAN_30_CORE_STATUS_INIT Initialized CAN controller. CAN_30_CORE_STATUS_INCONSISTENT STOP or SLEEP are inconsistent over common CAN controllers. CAN_30_CORE_DEACTIVATE_CONTROLLER RAM check failed CAN controller is deactivated. CAN_30_CORE_STATUS_WARNING WARNING state. CAN_30_CORE_STATUS_PASSIVE PASSIVE state. CAN_30_CORE_STATUS_BUSOFF BUSOFF mode. CAN_30_CORE_STATUS_SLEEP SLEEP mode.	

continues on next page

Table 4.16 – continued from previous page

Functional Description
Delivers the status of the hardware of the CAN controller.
Particularities
Feature has to be configured.
Options
Callcontext: ANY Reentrant: TRUE Synchronous: TRUE

Table 4.16 Can_30_Core_GetStatus

4.1.17 Can_30_Core_GetControllerErrorState

Prototype	
Std_ReturnType **Can_30_Core_GetControllerErrorState**(uint8 Controller, Can_ErrorStatePtrType ErrorStatePtr)	
Parameters	
Controller	CAN controller requested for status information.
ErrorStatePtr	Pointer to variable to store CAN controllers error state. Parameter must not be NULL and available for write access.
Return codes	
E_NOT_OK Controller state request has not been accepted. E_OK Controller state request has been accepted.	
Functional Description	
Delivers the CAN controllers error state.	
Particularities	
Feature has to be configured.	
Options	
Callcontext: ANY Reentrant: TRUE Synchronous: TRUE	

Table 4.17 Can_30_Core_GetControllerErrorState

4.1.18 Can_30_Core_GetControllerRxErrorCounter

Prototype	
Std_ReturnType **Can_30_Core_GetControllerRxErrorCounter**(uint8 Controller, Can_30_Core_ErrorCounterPtrType RxErrorCounterPtr)	
Parameters	
Controller	Requested CAN controller.

continues on next page

Table 4.18 – continued from previous page

RxErrorCounterPtr	Pointer to variable to store CAN controllers RX error counter. Parameter must not be NULL and available for write access.
Return codes	
E_NOT_OK Controller state request has not been accepted. E_OK Controller state request has been accepted.	
Functional Description	
Delivers the RX counter for the requested CAN controller.	
Particularities	
Feature has to be configured.	
Options	
Callcontext: ANY Reentrant: TRUE Synchronous: TRUE	

Table 4.18 Can_30_Core_GetControllerRxErrorCounter

4.1.19 Can_30_Core_GetControllerTxErrorCounter

Prototype	
Std_ReturnType **Can_30_Core_GetControllerTxErrorCounter**(uint8 Controller, Can_30_Core_ErrorCounterPtrType TxErrorCounterPtr)	
Parameters	
Controller	Requested CAN controller.
TxErrorCounterPtr	Pointer to variable to store CAN controllers TX error counter. Parameter must not be NULL and available for write access.
Return codes	
E_NOT_OK Controller state request has not been accepted E_OK Controller state request has been accepted	
Functional Description	
Delivers the TX counter for the requested CAN controller.	
Particularities	
Feature has to be configured.	
Options	
Callcontext: ANY Reentrant: TRUE Synchronous: TRUE	

Table 4.19 Can_30_Core_GetControllerTxErrorCounter

4.1.20 Can_30_Core_DisableControllerInterrupts

Prototype	
void **Can_30_Core_DisableControllerInterrupts**(uint8 Controller)	
Parameters	
Controller	The requested CAN controller.
Return codes	
No return value (void)	
Functional Description	
Disables the CAN controller interrupts.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.20 Can_30_Core_DisableControllerInterrupts

4.1.21 Can_30_Core_EnableControllerInterrupts

Prototype	
void **Can_30_Core_EnableControllerInterrupts**(uint8 Controller)	
Parameters	
Controller	The requested CAN controller.
Return codes	
No return value (void)	
Functional Description	
Enables the CAN controller interrupts.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.21 Can_30_Core_EnableControllerInterrupts

4.1.22 Can_30_Core_MainFunction_Mode

Prototype
void **Can_30_Core_MainFunction_Mode**( )
Parameters
None
Return codes
No return value (void)
Functional Description

continues on next page

Table 4.22 – continued from previous page

Polls the mode changes for all the CAN controllers asynchronous if not accomplished in Can_30_Core_SetControllerMode().
Particularities
Must not be called nested and must not be interrupted by other CAN main function calls.
Options
Callcontext: TASK Reentrant: FALSE Synchronous: TRUE

Table 4.22 Can_30_Core_MainFunction_Mode

4.1.23 Can_30_Core_MainFunction_BusOff

Prototype
void **Can_30_Core_MainFunction_BusOff**( )
Parameters
None
Return codes
No return value (void)
Functional Description
Polling of bus off events to accomplish the bus off handling for all CAN controllers.
Particularities
Must not be called nested and must not be interrupted by other CAN main function calls.
Options
Callcontext: TASK Reentrant: FALSE Synchronous: TRUE

Table 4.23 Can_30_Core_MainFunction_BusOff

4.1.24 Can_30_Core_MainFunction_Read

Prototype
void **Can_30_Core_MainFunction_Read**( )
Parameters
None
Return codes
No return value (void)
Functional Description
Calls the polling tasks of the extensions and handles the RxQueue.
Particularities
Must not be called nested and must not be interrupted by other CAN main function calls.
Options

continues on next page

Table 4.24 – continued from previous page

Callcontext: TASK Reentrant: FALSE Synchronous: TRUE

Table 4.24 Can_30_Core_MainFunction_Read

4.1.25 Can_30_Core_MainFunction_Write

Prototype
void **Can_30_Core_MainFunction_Write**( )
Parameters
None
Return codes
No return value (void)
Functional Description
Calls the polling tasks of the extension and handles Tx Confirmation.
Particularities
Must not be called nested and must not be interrupted by other CAN main function calls.
Options
Callcontext: TASK Reentrant: FALSE Synchronous: TRUE

Table 4.25 Can_30_Core_MainFunction_Write

4.1.26 Can_30_Core_MainFunction_Wakeup

Prototype
void **Can_30_Core_MainFunction_Wakeup**( )
Parameters
None
Return codes
No return value (void)
Functional Description
Polling WAKEUP events for all CAN controllers to accomplish the WAKEUP handling.
Particularities
Must not be called nested and must not be interrupted by other CAN main function calls.
Options
Callcontext: TASK Reentrant: FALSE Synchronous: TRUE

Table 4.26 Can_30_Core_MainFunction_Wakeup

4.2 Required Functions

4.2.1 Functions used by module

The module uses these functions from other modules. Refer to their documentation for API details.

Module	API

CanIf	CanIf_CancelTxNotification() (non AUTOSAR)
	CanIf_TxConfirmation()
	CanIf_CancelTxConfirmation()
	CanIf_RxIndication()
	CanIf_ControllerBusOff()
	CanIf_ControllerModeIndication()
	CanIf_ControllerErrorStatePassive()
	CanIf_ErrorNotification()

SchM	SchM_Enter_<Can_30_Core_CAN_30_CORE_EXCLUSIVE_AREA>() SchM_Exit_<Can_30_Core_CAN_30_CORE_EXCLUSIVE_AREA>()
VStdLib	VStdLib_MemCpy()

OS	OS_TICKS2MS_<counterShortName>
	GetElapsedValue()
	GetCounterValue()

EcuM	EcuM_CheckWakeup()
	EcuM_BswErrorHook()

Det (optional)	Det_ReportError
Application (optional)	see section *Callout functions*

Rtm (optional)	Rtm_Start()
	Rtm_Stop()

Table 4.27 Required Functions

4.2.2 Callout functions

The module requires these callout functions to be provided by the user code:

4.2.2.1 Appl_30_CoreCanTimerStart

Prototype	
void **Appl_30_CoreCanTimerStart**(uint8 Controller, uint8 Source)	
Parameters	
Controller	CAN controller on which the hardware observation takes place.
Source	Source for the hardware observation.
Return codes	
No return value (void)	
Functional Description	
Service function to start an observation timer.	
Particularities	
Feature has to be configured.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.28 Appl_30_CoreCanTimerStart

4.2.2.2 Appl_30_CoreCanTimerEnd

Prototype	
void **Appl_30_CoreCanTimerEnd**(uint8 Controller, uint8 Source)	
Parameters	
Controller	CAN controller on which the hardware observation takes place.
Source	Source for the hardware observation.
Return codes	
No return value (void)	
Functional Description	
Service function to end an observation timer.	
Particularities	
Feature has to be configured.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.29 Appl_30_CoreCanTimerEnd

4.2.2.3 Appl_30_CoreCanTimerLoop

Prototype	
Can_ReturnType **Appl_30_CoreCanTimerLoop**(uint8 Controller, uint8 Source)	
Parameters	
Controller	CAN controller on which the hardware observation takes place.
Source	Source for the hardware observation.
Return codes	
CAN_NOT_OK When loop shall be broken (observation stops). CAN_NOT_OK should only be used in case of a time out occurs due to a hardware issue. After this an appropriate error handling is needed (see chapter *Hardware Loop Check / Timeout Monitoring*). CAN_OK When loop shall be continued (observation continues).	
Functional Description	
Service function to check (against generated maximum loop value) whether a hardware loop shall be continued or broken.	
Particularities	
Feature has to be configured.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.30 Appl_30_CoreCanTimerLoop

4.2.2.4 Appl_30_Core_GenericConfirmation (API1)

Prototype	
Can_ReturnType **Appl_30_Core_GenericConfirmation**(uint16 PduId)	
Parameters	
PduId	Handle of the transmitted PDU.
Return codes	
CAN_OK Higher layer confirmation will be called afterwards (CanIf_TxConfirmation()). CAN_NOT_OK Higher layer confirmation will not be called afterwards.	
Functional Description	
Common TX notification callback that will be called before PDU specific callback will be called. Application callback function which informs about Tx PDU being sent to the CAN bus. It can be used to block confirmation or route the information to other layer as well.	
Particularities	
Feature has to be configured.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.31 Appl_30_Core_GenericConfirmation

4.2.2.5 Appl_30_Core_GenericConfirmation (API2)

Prototype	
Can_ReturnType **Appl_30_Core_GenericConfirmation**(uint8 Controller, Can_PduInfoPtrType DataPtr)	
Parameters	
Controller	CAN controller which send the PDU.
DataPtr	Pointer to a Can_PduType structure including CAN-Id (containing IDE,FD bit), DataLength, PDU and data pointer (read only).
Return codes	
CAN_OK Higher layer confirmation will be called afterwards (CanIf_TxConfirmation()). CAN_NOT_OK Higher layer confirmation will not be called afterwards.	
Functional Description	
Common TX notification callback that will be called before PDU specific callback will be called. Application callback function which informs about Tx PDU being sent to the CAN bus. It can be used to block confirmation or route the information to other layer as well.	
Particularities	
> Feature has to be configured. > A new transmission within this call out will corrupt the DataPtr context, if the CanHardwareObject is a FIFO object and the FIFO is full.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.32 Appl_30_Core_GenericConfirmation

4.2.2.6 Appl_30_Core_GenericPrecopy

Prototype	
Can_ReturnType **Appl_30_Core_GenericPrecopy**(uint8 Controller, Can_IdType ID, uint8 DataLength, Can_DataPtrType DataPtr)	
Parameters	
Controller	CAN controller which received the PDU.
ID	CAN-Id of the received PDU (including IDE,FD). In case of extended or mixed CAN-Id systems the highest bit (bit 31) is set to mark an extended CAN-Id. FD-bit (bit 30) can be masked out with a define in the user config file.
DataLength	Data length of the received PDU.
DataPtr	Pointer to the data of the received PDU (read only).
Return codes	
CAN_OK Higher layer indication will be called afterwards (CanIf_RxIndication()). CAN_NOT_OK Higher layer indication will not be called afterwards.	
Functional Description	

continues on next page

Table 4.33 – continued from previous page

Common RX indication callback that will be called before PDU specific callback will be called. Application callback function which informs about all incoming Rx PDUs including the contained data. It can be used to block notification to upper layer. E.g. to filter incoming PDUs or route it for special handling.
Particularities
> Feature has to be configured. > "DataPtr" is read only and must not be accessed for further write operations (explicitly not const because of backward compatibility). > The CAN protocol allows the usage of data length values greater than eight (CAN-FD). Depending on the implementation of this callback it may be necessary to consider this special case (e.g. if the data length is used as index value in a buffer write access).
Options
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE

Table 4.33 Appl_30_Core_GenericPrecopy

4.2.2.7 Appl_30_Core_GenericPreTransmit

Prototype	
void **Appl_30_Core_GenericPreTransmit**(uint8 Controller, Can_PduInfoVarPtrType PduPtr)	
Parameters	
Controller	CAN controller on which the PDU will be send.
PduPtr	Pointer to a Can_PduType structure including CAN-Id (containing IDE,FD bit), DataLength, PDU and data pointer.
Return codes	
No return value (void)	
Functional Description	
Application callback function allowing the modification of the data to be transmitted (e.g.: add CRC).	
Particularities	
> Feature has to be configured. > Do not extend PduPtr->length beyond the maximum length for the given PduPtr->swPduHandle (see related CanHardwareObject\CanMaxDataLen). > Do not write beyond the PduPtr->length to the PduPtr->sdu.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.34 Appl_30_Core_GenericPreTransmit

4.2.2.8 Appl_30_Core_CanCorruptMailbox

Prototype	
void **Appl_30_Core_CanCorruptMailbox**(uint8 Controller, Can_HwHandleType mailboxHandle)	
Parameters	
Controller	CAN controller for which the check failed.
mailboxHandle	Hardware handle of the defect CanHardwareObject.
Return codes	
No return value (void)	
Functional Description	
Will notify the application (during re-initialization) about a defect CanHardwareObject within the CAN Hardware Unit.	
Particularities	
Feature has to be configured.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.35 Appl_30_Core_CanCorruptMailbox

4.2.2.9 Appl_30_Core_CanRamCheckFailed

Prototype	
uint8 **Appl_30_Core_CanRamCheckFailed**(uint8 Controller)	
Parameters	
Controller	CAN controller for which the check failed.
Return codes	
CAN_DEACTIVATE_CONTROLLER Deactivate the CAN controller. CAN_ACTIVATE_CONTROLLER Activate the CAN controller.	
Functional Description	
Will notify the application (during re-initialization) about a defect CAN controller due to a previous failed mailbox check. The return value decides how to proceed with the initialization.	
Particularities	
Feature has to be configured.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.36 Appl_30_Core_CanRamCheckFailed

4.2.2.10 Appl_30_CoreCanInterruptDisable

Prototype	
void **Appl_30_CoreCanInterruptDisable**(uint8 Controller)	
Parameters	
Controller	CAN controller for the CAN interrupt lock.
Return codes	
No return value (void)	
Functional Description	
Disabling of CAN Interrupts by the application. E.g.: The CAN driver itself should not access the common Interrupt Controller due to application specific restrictions (like security level). Or the application likes to be informed because of an CAN interrupt lock.	
Particularities	
Feature has to be configured.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.37 Appl_30_CoreCanInterruptDisable

4.2.2.11 Appl_30_CoreCanInterruptRestore

Prototype	
void **Appl_30_CoreCanInterruptRestore**(uint8 Controller)	
Parameters	
Controller	CAN controller for the CAN interrupt unlock.
Return codes	
No return value (void)	
Functional Description	
Re-enabling of CAN Interrupts by the application. E.g.: The CAN driver itself should not access the common Interrupt Controller due to application specific restrictions (like security level). Or the application likes to be informed because of an CAN interrupt lock.	
Particularities	
Feature has to be configured.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.38 Appl_30_CoreCanInterruptRestore

4.2.2.12 Appl_30_Core_CanOverrun

Prototype	
void **Appl_30_Core_CanOverrun**(uint8 Controller)	
Parameters	
Controller	CAN controller for which the overrun was detected.
Return codes	
No return value (void)	
Functional Description	
Called when an overrun is detected for a BasicCAN. Alternatively a DET call can be selected instead of ("CanOverrunNotification" is set to "DET").	
Particularities	
Feature has to be configured.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.39 Appl_30_Core_CanOverrun

4.2.2.13 Appl_30_Core_CanFullCanOverrun

Prototype	
void **Appl_30_Core_CanFullCanOverrun**(uint8 Controller)	
Parameters	
Controller	CAN controller for which the overrun was detected.
Return codes	
No return value (void)	
Functional Description	
Called when an overrun is detected for a FullCAN. Alternatively a DET call can be selected instead of ("CanOverrunNotification" is set to "DET").	
Particularities	
Feature has to be configured.	
Options	
Callcontext: ANY Reentrant: FALSE Synchronous: TRUE	

Table 4.40 Appl_30_Core_CanFullCanOverrun

4.3 Service Ports

This module does not provide any service ports.

5 Configuration

The functionalities of the module are configured with DaVinci Configurator Classic.

For details on configuration options refer to the module-specific description that is shown in DaVinci Configurator Classic.

5.1 Configuration Variants

This module supports these configuration variants:

- VARIANT-PRE-COMPILE
- VARIANT-POST-BUILD-LOADABLE
- VARIANT-POST-BUILD-SELECTABLE

The configuration classes of the parameters depend on the supported configuration variants. For their definitions please see the module-specific BSWMD ARXML file.

The configuration of Post-Build Loadable is describe in *[5]*.

5.2 General Configuration Hints

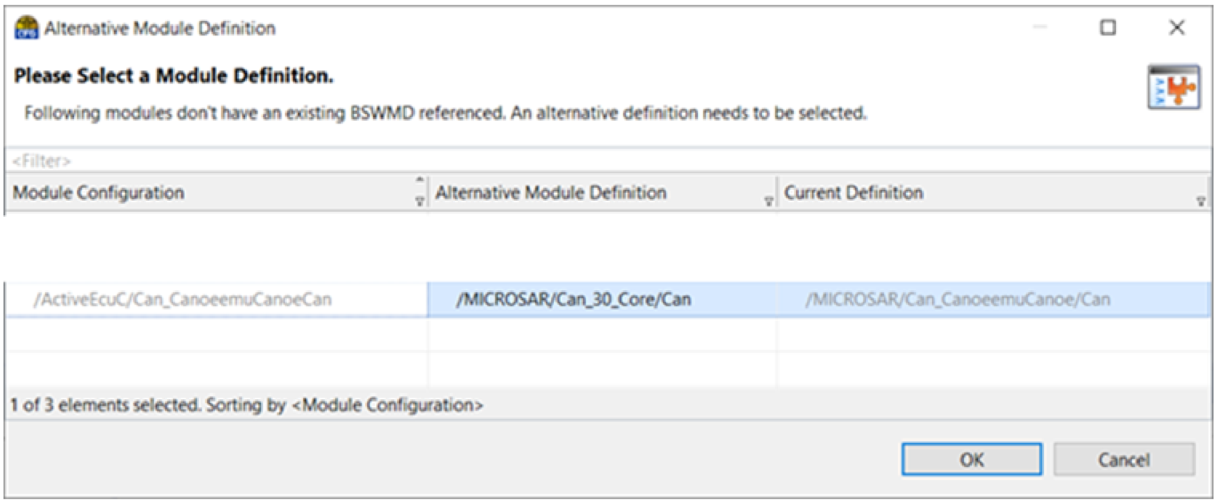
The CAN Driver consists of two separate modules in the Configurator (see also *chapter 1.1*). There is a module 'Can' and an additional module is called 'vCan'; the 'Can' module is responsible for all CAN Hardware Unit independent behavior and configuration, whereas the 'vCan' module contains CAN Hardware Unit specific configuration as an extension to 'Can'. It is also possible to add multiple 'vCan' modules to support different CAN Hardware Units within one configuration.

Note
To register a CAN Hardware Unit specific vCan module at the 'Can' (Core) module it is necessary to fill out the parameter `/MICROSAR/Can_30_Core/Can/CanConfigSet/CanController/CanvCanExtensionRef`.
A Validator within the configuration tool will provide support to add and refer the needed vCan module.

5.3 Configuration Migration

When updating from a configuration which was not created by using this CAN driver (`/MICROSAR/Can_30_Core/`), the generation tool will ask to select a "Alternative Module Definition" instead of the "Current Definition". Where the "Alternative Module" is your new `/MICROSAR/Can_30_Core/` CAN driver instead of the previously used CAN driver.

As an example (see below), select and confirm your CAN driver to be migrated.


Alternative Module Definition

Please Select a Module Definition.

Following modules don't have an existing BSWMD referenced. An alternative definition needs to be selected.

<Filter>

Module Configuration	Alternative Module Definition	Current Definition
/ActiveEcuC/Can_CanoeemuCanoeCan	/MICROSAR/Can_30_Core/Can	/MICROSAR/Can_CanoeemuCanoe/Can

1 of 3 elements selected. Sorting by <Module Configuration>

Figure 5.1 Can Driver selection

The related `vCan` module (see also see also *chapter 1.1*) will be added automatically.

A migration script will adapt the `/MICROSAR/Can_30_Core/` settings where needed.

There will be some `vCan` module parameters (like CAN cell base address), which must be set manual.

6 Glossary and Abbreviations

6.1 Glossary

Term	Description
HighEnd (license)	Product license to support an extended feature set.
vCan	The hardware specific part of the CAN driver represents the specific CAN Hardware Unit abstraction as an extension to the 'Can' module. It is implemented as a separate module with own static code and generator. There could be multiple vCan's if the hardware contains multiple CAN Hardware Units.

6.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
Appl	Application
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
BSWMD	Basis Software Module Description
CAN	Controller Area Network
CanIf	Can Interface module
DET	Development Error Tracer
DLC	Data Length Code
ECU	Electronic Control Unit
EcuM	ECU State Manager module
FD	Flexible Data-rate
FIFO	First In First Out
HIS	Hersteller Initiative Software
HW	Hardware
MICROSAR Classic	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
OS	Operating System module
PDU	Protocol Data Unit
RTM	Runtime Measurement module
VStdLib	Vector Standard Library

Table 6.2 Abbreviations

7 Contact

Visit our website for more information on

> News
> Products
> Demo software
> Support
> Training data
> Addresses

www.vector.com