Technical Reference

Version 2.10.01

	Thomas Arnold, Rüdiger Naas, Eugen Stripling
	2.10.01
	Released

Thomas Arnold	2006-06-22	1.0	Initial version
Thomas Arnold	2006-07-05	1.1	Minor corrections (Review) Add additional DET error codes
Thomas Arnold	2006-07-05	1.2	Add justification for possible compiler warning
Thomas Arnold	2006-10-30	1.3	Add additional features (TxFullCAN, TxPolling, …), Add GENy configuration chapter.
Hartmut Hörner	2007-01-04	1.4	Added information about supported AUTOSAR version
Thomas Arnold	2007-01-19	1.5	Add additional features (BusOff polling, Post build configuration). Changes in GENy configuration chapter.
Thomas Arnold	2007-06-04	1.6	Adapt to AUTOSAR 2.1
Thomas Arnold	2007-07-20	1.7	Switch to new template / modifications for Autosar 2.1
Thomas Arnold	2008-03-03	1.8	Add Extended ID support
Thomas Arnold	2008-03-10	2.0	Adapt to AUTOSAR 3
Thomas Arnold	2008-05-16	2.1	Changes due to review: - Add info about DLC to ReadRxPduData API - Layout changes - Remove Can_MainFunction API - …
Thomas Arnold	2008-06-11	2.2	Add CanIf_CanTrcv.h to chapter 4.1.2
Thomas Arnold	2008-08-04	2.3	Add description of WakeUpValidation (Chapters 3.12, 3.15, 6.19) Update GENy Screenshots and description (Chapter 5)
Thomas Arnold	2008-10-07	2.4	Change GENy attribute names (Chapter 5) Update include structure (Chapter 4.2)

Thomas Arnold	2008-10-17	2.5	Add description of AUTOSAR 2.1 ComM support (Chapter 3.16)
Thomas Arnold	2008-10-31	2.6	Update of figure 3-1 Update GENy screenshots / attribute names (Chapter 5) Rework chapter 6 API description Minor improvements
Rüdiger Naas	2009-06-29	2.7	Description Double Hash search algorithm added
Rüdiger Naas	2009-08-25	2.7.1	Limitation for API CanIf_Transmit() added
Rüdiger Naas	2009-09-25	2.7.2	Chapter Deviations/Limitations added
Rüdiger Naas	2009-11-23	2.7.3	Defines for CanIf_PduSetModeType changed Example for how to convert Upper/Lower ID to mask and code.
Rüdiger Naas	2010-01-11	2.7.4	Minor changes regarding indication function types.
Rüdiger Naas	2010-03-08	2.8.0	Dynamic transmit L-PDU handles EcuM_GeneratorCompatibilityError API added
Rüdiger Naas	2010-06-30	2.9.0	Expansion of the description for the interrupt lock mechanism Typo corrected for chapter “sleep/wakeup” Tx buffer handling expansion Postbuild parameter description changed Bit Queue support
Rüdiger Naas	2011-01-12	2.10.0	Some typos corrected at chapter “sleep/wakeup”.
Eugen Stripling	2011-06-30	2.10.01	DLC check against not optimized DLC

Table 1-1 History of the Document

[1]	AUTOSAR_SWS_CAN_Interface.pdf	3.0.1
[2]	AUTOSAR_SWS_DET.pdf	2.2.0
[3]	AUTOSAR_SWS_DEM.pdf	2.2.1
[4]	AUTOSAR_BasicSoftwareModules.pdf	1.2.0

Table 1-2 References Documents

⚠	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.

This document describes the functionality, API and configuration of the AUTOSAR CAN Interface as specified in [1]. It is based on the AUTOSAR specification release 3. The CAN Interface is a hardware independent layer with a standardized interface to the CAN Driver and CAN Transceiver Driver layer and upper layers like PDU Router, Communication Manager and the Network Management.

	3	
	pre-compile, link-time, post-build	
	CANIF_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
	CANIF_MODULE_ID	60 (according to ref. [4])

* For the precise AUTOSAR Release 3.x please see the release specific documentation.

The following figure shows where the CAN Interface is located in the AUTOSAR architecture.

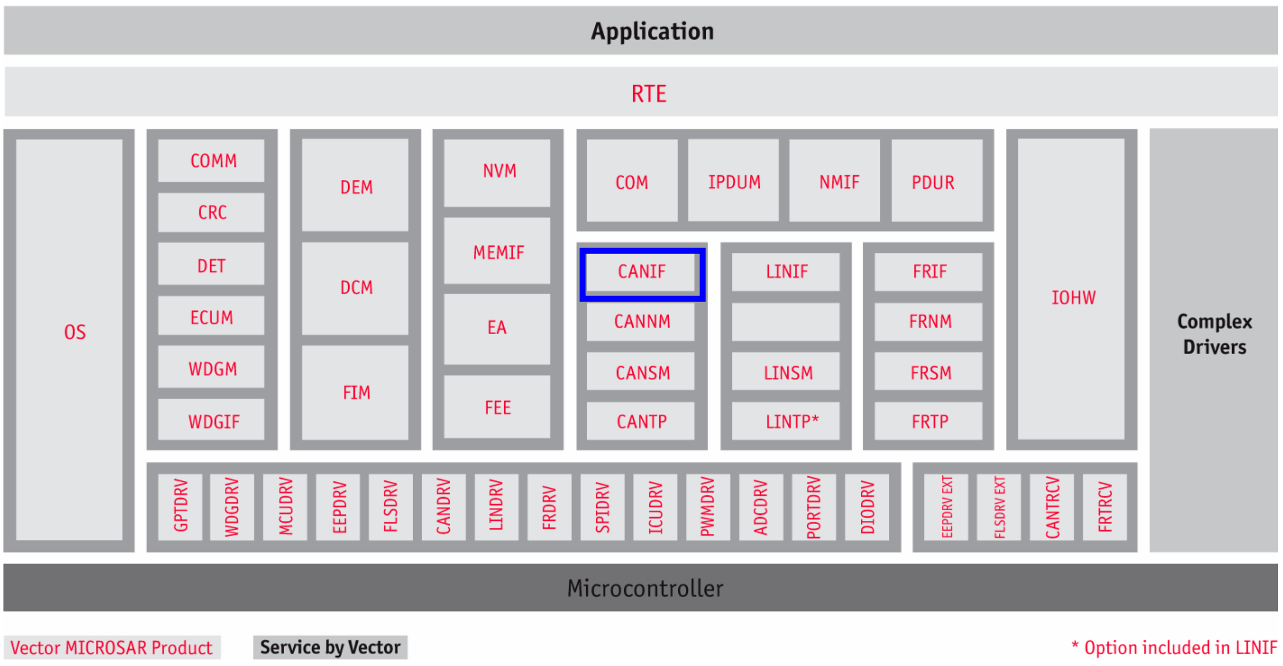


Figure 2-1 AUTOSAR layer model

The CAN Interface provides a standardized interface for all upper layers which require CAN communication. Therefore these upper layers have to communicate with the CAN

Interface which is responsible for the CAN communication. This includes transmission and reception of messages as well as state handling of the CAN controllers.

The next figure shows the interfaces to adjacent modules of the CAN Interface. These interfaces are described in chapter 6.

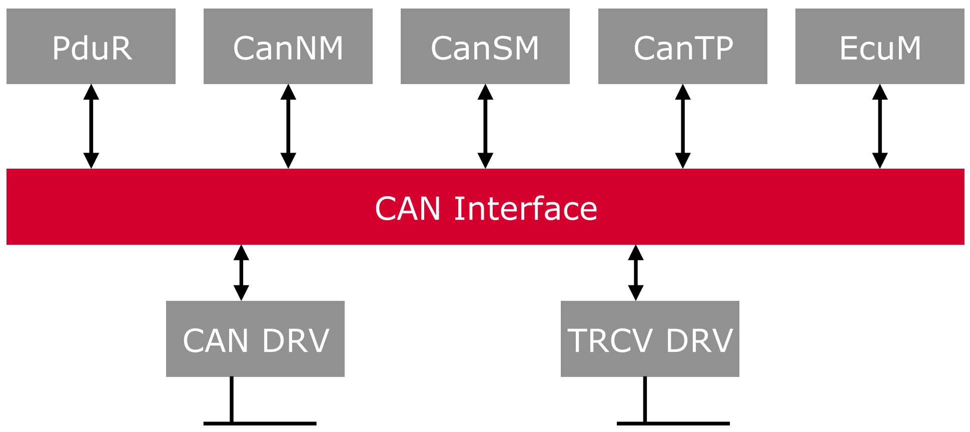


Figure 2-2 Interfaces to adjacent modules of the CAN Interface

Please note that the CAN Interface is tailored by Vector Informatik according to customer requirements before delivery. As a result not all features listed below might be supported by a delivered module.

For deviations and extensions regarding the AUTOSAR standard [1], please see chapter 7.

Generic Initialization	■	General initialization of the CAN Interface (CanIf_Init())
Controller specific Initialization	■	CanIf_ControllerInit()
Transmission	■	Transmission of PDUs
Dynamic transmission	■	Transmission of PDUs with changeable CAN IDs
Transmit buffer	■	Buffering of PDUs (send request and data) in the CAN Interface
Cancellation of Tx PDUs	■	Cancellation of PDUs and requeueing. (Feature to avoid inner priority inversion)
Transmit confirmation	■	Call back for successful transmission
Reception	■	Reception of PDUs
Receive indication	■	Call back for reception of PDUs
DLC check	■	Check DLC of received PDUs against predefined values
Sleep mode	■	Controller support sleep mode
External wake up (CAN)	■	Support external wake up by CAN Driver
External wake up (Transceiver)	■	Support external wake up by Transceiver Driver
Wake up validation	■	Support wake up validation for external wake up events
Internal wake up	■	Internal wake up by calling CanIf_SetControllerMode()
Stop mode	■	Controller support stop mode
BusOff detection	■	Handling of bus off notifications
DEM	■	Support Diagnostic Event Manager (error notification)
DET	■	Support Development Error Detection (error notification)
Tx BasicCAN	■	Standard mailbox to send CAN frames (Used by CAN Interface data queue)

Tx FullCAN	■	Separate mailbox for special Tx message used
Rx BasicCAN	■	Standard mailbox to receive CAN frames (depending on hardware, FIFO or shadow buffer supported)
Rx FullCAN	■	Separate mailbox for special Rx message used
Transceiver handling	■	API for upper layers to set and read transceiver states; Interface to the Transceiver Driver
Version API	■	API to read out component version
Supported ID types		
- Standard Identifiers	■	Support of CAN Standard (11 bits) identifiers
- Extended Identifiers	■	Support of CAN Extended (29 bits) identifiers
- Mixed Identifiers	■	Support standard as well as extended identifiers
Multiple CAN networks	■	Each CAN network has to be connected to exactly one controller

Table 3-1 List of supported features

Several functions are available to initialize the CAN Interface. The following code example shows which functions have to be called to initialize the CAN Interface and to allow transmission and reception.

```
CanIf_InitMemory();/* Optional call which reinitializes global
                   variables to set the CAN Interface back to
                   uninitialized state. */

(CanTrcv_xxx_InitMemory() and) CanTrcv_xxx_Init

                   /* have to be called to initialize the CAN
                   Transceiver Driver and set the CAN Transceiver
                   to the preconfigured state. For some CAN
                   Controllers it is necessary to have a recessive
                   signal on the Rx Pin to be able to initialize
                   the CAN Controller. This means the transceiver
                   has to be set to "normal mode" before
                   CanIf_Init() is called. */

(Can_InitMemory() and )Can_Init();

                   /* have to be called before CanIf_Init is
                   called. */

CanIf_Init(<PtrToCanIfConfiguration>);

                   /* Global initialization of the CAN Interface,
                   all available controllers and then CAN Driver
                   are initialized within this call. If selectable
                   post build configuration is active a valid
                   configuration has to be passed to the call of
                   CanIf_Init. In other cases the parameter will be
                   ignored and a NULL pointer can be used */
CanIf_SetControllerMode(0, CANIF_CS_STARTED);
                   /* The controller mode for controller 0 is set
                   to started mode. This means the CAN controller
                   is initialized and ready to communicate
                   (acknowledge of the CAN controller is
                   activated). Communication is not yet possible
                   because the CAN Interface will neither pass Tx
                   PDUs from higher layers to the CAN Driver nor
                   accept Rx PDUs from the CAN Driver. */
CanIf_SetPduMode(0, CANIF_SET_ONLINE);
                   /* The PDU mode in the CAN Interface is switched
                   to online mode. After initialization this mode
                   remains in the state CANIF_GET_OFFLINE until the
                   CanIf_SetPduMode function is called. Now
                   transmission requests will be passed from the
                   upper layer to the CAN Driver and Rx PDUs are
                   forwarded from the CAN Driver to the
                   corresponding higher layer. */
```

The transmission of PDUs is only possible after the CAN Interface and CAN Driver is initialized and the CAN Interface resides in the CANIF_CS_STARTED / CANIF_GET_ONLINE or CANIF_CS_STARTED / CANIF_GET_TX_ONLINE mode. In all other states the Tx requests are rejected by the CAN Interface.

The Tx request has to be initiated by a call to the function:

```
CanIf_Transmit(<TxPduId>, <PduInfoPtr>);
```

The CAN Interface uses the PDU ID to acquire more information to transmit the message from the generated data. This data is used to call the CAN Driver's Can_Write function which needs information about the PDU like ID, length, data and about the hardware transmit handle which represents the mailbox used for transmission of the PDU.

After the message was successfully transmitted on the bus a confirmation function will be called by the CAN Driver either from interrupt context or in case of Tx polling from task context. This confirmation is dispatched in the CAN Interface to notify the corresponding higher layer about the transmission of the PDU. For this purpose for each PDU a call back function has to be specified at configuration time.

The transmission request will be rejected by returning E_NOT_OK in the following cases:

- The CAN Interface is not in the controller state CANIF_CS_STARTED

- The CAN Interface is not in the channel mode CANIF_GET_ONLINE or CANIF_GET_TX_ONLINE
- The transmit buffer is not active and the corresponding mailbox used for transmission is occupied (BasicCAN Tx messages only).
- An error occurred during transmission (DET or DEM will be informed)

The feature is activated by the global switch “Dynamic Tx Objects”.

The adjustments for the dynamic objects are the same as for the static with the exception that the CAN ID and the attribute for ext./std. ID can be selected manually.

For default the dynamic object has the CAN ID parameterized during configuration time until it will be change by the `CanIf_SetDynamicTxId()` API. If a extended ID is written by the API, the most significant bit must be set.

The PDU IDs of the dynamic objects are represented as symbolic names for the file canif_cfg.h.

The CAN Interface provides a mechanism to buffer one Tx request including data for each BasicCAN PDU. This means if the BasicCAN Tx hardware objects are occupied each PDU configured to be transmitted via this hardware object can be stored in the CAN Interface until the hardware transmit object is free again.

If a specific PDU is resent and the Tx buffer for this PDU is already in use, the data stored for this PDU will be overwritten in the CAN Interface to make sure the newest data will be transmitted.

The entries stored in the Tx buffer will be processed from a Tx confirmation interrupt or from the CAN Driver’s Tx main function in polling mode.

Within the confirmation function the list of stored PDUs in the transmit buffer will be searched for the PDU with highest priority (lowest CAN identifier) which will be the first to be removed from queue and written to the hardware by the CAN Driver.

If the CAN Controller supports multiple hardware objects the Tx Buffer will be used to avoid inner priority inversion. This means if the CAN Interface passes a transmit request to the CAN Driver while all Tx hardware message objects are occupied (at least one message object is occupied by a CAN message with lower priority than the message used for the current transmit request) the CAN Driver will initiate a cancellation of the message with the lowest priority. The cancelled message will be stored in the Tx buffer if the Tx buffer is free, otherwise it will be discarded to make sure the newest data will be transmitted. A Tx hardware message object will be free due to the cancellation and allows the CAN Interface to pass the message with the highest priority to the CAN Driver.

Reception of PDUs is only possible in the state CANIF_CS_STARTED / CANIF_GET_ONLINE or CANIF_CS_STARTED / CANIF_GET_RX_ONLINE. In all other states the PDUs received by the CAN Driver will be discarded in the CAN Interface without notification of the upper layers.

The CAN Interface supports FullCAN as well as BasicCAN reception. The upper layers won't notice any differences between these two reception types as in both cases a call back function will be called which was configured for the specified PDU in the generation tool.

The upper layer will be notified about the PDU ID, the received data and depending on the used indication function about the length of the received data. The PDU IDs are zero based lists which will be distributed for each call back function. This means a specific Rx PDU ID can occur multiple times in the CAN Interface and has to be evaluated in the corresponding indication function by the higher layer.

ℹ	The PDU IDs of Rx messages are not unique. The assigned indication function and PDU ID have to be used to identify one specific Rx PDU.

In case of BasicCAN reception the CAN Interface has to search through a list of all known Rx messages and compare the received ID with the ID in the Rx message list.

The CAN Interface offers two different search algorithms:

Linear search: The list of all Rx PDUs is searched from high priority (Low CAN Identifier) to low priority (High CAN Identifier). This algorithm is efficient for a small amount of Rx messages.

Double Hash search: The Rx PDU is calculated via two special hash functions. The algorithm is very efficient for a high amount of Rx messages and always takes the same time.

Binary Search: The list of Rx PDUs is split in two equal sized parts and the search is continued recursively on a list of PDUs which contains half the messages. This search algorithm terminates faster for big amounts of Rx messages than the linear search.

⚠	The binary search algorithm cannot be used for mixed ID systems.

The BasicCAN message object can be used to receive groups of Rx PDUs called ranges. A range is defined by a mask and a code which define a group of messages using the following expression:

<Received ID> & <Mask> == <Code>

One PDU ID is assigned to all messages in the range; this means the upper layer won't be able to get additional message properties like CanID, …

For each range an indication function can be assigned in the generation tool, which is used to notify the higher layer about the reception of a message.

For the definition of ranges with upper and lower ID a conversion to Mask and Code is necessary. The example below helps you how to do this.

Lower ID: 0x400

Upper ID: 0x43F

The Code is same as the lower ID:

Code = 0x400

You need the Count which is upper ID – lower ID -> 0x43F – 0x400 = 0x3F

The Count 0x3F makes 000 0011 1111b in 11bit binary format. For a range with extended IDs the Count needs to be 29 bit wide.

The Mask is calculated out of negated Count and a 11 Bit mask:

Mask = ~0x3F & 0x7FF = 0x7C0

For extended IDs you need a 29 Bit mask:

Mask = ~0x3F & 0x1FFF FFFF = 0x1FFF FFC0

Note:

If for Count the first set bit is followed by unset bits on lower significant positions, for the calculation of the Mask these bits need to be set. For example a Count of 0xA3 (1010 0011b) you need to calculate with the Count 0xFF (1111 1111b). The consequence is that more IDs are received as intended.

A DLC check will be executed for all received messages after they passed the search algorithm (PDU is in Rx list) or if they are defined to be received in FullCAN message objects. The DLC check has to be enabled at configuration time (only pre-compile configuration). If the DLC check has been activated at pre-compile time the DLC check can be disabled for single Rx PDUs and for ranges at post-build time. Refer to chapter 5.2 Channel specific properties and chapter 5.5 Rx message properties for configuration details.

The DLC check will verify if the received DLC is greater or equal to the DLC specified at configuration time. If the DLC is less then the configured one a DEM Error will be raised and the reception of the PDU is abandoned.

Described DLC check is usually performed against optimized DLC from database. That means that data bytes within a CAN message which are not aligned by signals are not taken into account of calculation of DLC which is used for DLC check. If this optimization is not desired this can be disabled via checkbox *Dlc Check Optimization* (s. Figure 5-3). The DLC check verifies if the DLC of a received CAN message is greater or equal to the configured DLC. If the received DLC is less than the configured one a DEM error will be raised and the reception of the PDU is abandoned.

The CAN Interface knows two main types of communication modes.

The controller mode represents the physical state of the CAN controller. The following modes are available:

- CANIF_CS_STOPPED
- CANIF_CS_STARTED
- CANIF_CS_SLEEP
- CANIF_CS_UNINIT

There is no state called bus off. Bus off is treated as a transition from STARTED to STOPPED mode. All transitions have to be initiated using the API function CanIf_SetControllerMode(Network, RequestMode). The controller mode can be switched for each controller independent of the state of other controllers in the system.

The state CANIF_CS_UNINIT is left after CanIf_InitController(Network) is called and can only be entered by a reset of the ECU.

The modes CANIF_CS_SLEEP and CANIF_CS_STARTED can only be entered from CANIF_CS_STOPPED. This means a transition from STARTED to SLEEP and vice versa is not possible without requesting the STOPPED mode first.

It is always possible to request the currently active controller mode with the API function CanIf_GetControllerMode(Network, ControllerModePtr).

The other type of communication mode is completely processed by software (it does not represent any state of the hardware). Transitions of the channel mode are only possible if the controller mode is set to CANIF_CS_STARTED. In all other controller modes the channel mode is automatically set to CAN_GET_OFFLINE and cannot be changed by a call to CanIf_SetChannelMode.

The following channel modes are available:

- CANIF_GET_OFFLINE
  Rx and Tx path is switched offline
- CANIF_GET_RX_ONLINE
  Rx path online, Tx path offline
- CANIF_GET_TX_ONLINE
  Rx path offline, tx path online
- CANIF_GET_ONLINE
  Rx and Tx path is switched online
- CANIF_GET_OFFLINE_ACTIVE
  Rx and Tx path offline, confirmation is emulated by the CAN Interface
- CANIF_GET_OFFLINE_ACTIVE_RX_ONLINE
  Rx path online, Tx path offline, confirmation is emulated be the CAN Interface

The channel modes have to be set with the API function CanIf_SetChannelMode(channel, mode) and can be requested with the function CanIf_GetChannelMode(channel, modePtr).

The CAN Interface can process events in polling and interrupt mode. As the polling of events is executed by other layers (e.g. Can Driver, Transceiver Driver) the CAN Interface is in all cases notified by call back functions which are called in different contexts.

	There is no need for changes in the configuration to run the CAN Interface in polling mode.

AUTOSAR specifies two mechanisms of error notification and reporting. Both mechanisms are supported by the CAN Interface and can be activated at configuration time (Pre-compile configuration).

Development errors are reported to DET using the service Det_ReportError().This feature is normally activated during the development phase to detect fatal errors in configuration and integration of the CAN Interface with other layers.

The reported CAN Interface ID is 60.

The reported service IDs identify the services which are described in 6. The following table presents the service IDs and the related services:

1	CanIf_Init
2	CanIf_InitController
3	CanIf_SetControllerMode
4	CanIf_GetControllerMode
5	CanIf_Transmit
6	CanIf_ReadRxPduData
9	CanIf_SetPduMode
10	CanIf_GetPduMode
11	CanIf_GetVersionInfo
13	CanIf_SetTransceiverMode
14	CanIf_GetTransceiverMode
15	CanIf_GetTrcvWakeupReason
16	CanIf_SetTransceiverWakeupMode
17	CanIf_CheckWakeup
18	CanIf_CheckValidation
19	CanIf_TxConfirmation
20	CanIf_RxIndication
21	CanIf_CancelTxConfirmation
22	CanIf_ControllerBusoff
250	CanIf_CancelTransmit
251	CanIf_CancelTxNotification

Table 3-2 Mapping of service IDs to services

The errors reported to DET are described in the following table:

10	`CANIF_E_PARAM_CANID`	The error code is used if an invalid CAN identifier is passed to the CAN Interface from the CAN driver during the reception of a RxPDU. The error can be raised from: -
11	`CANIF_E_PARAM_DLC`	The error will be reported by - if a DLC greater than 8 is passed to the CAN Interface during reception.
12	`CANIF_E_PARAM_LPDU`	The error will be raised by the following functions if an unexpected PduId is passed to the function by either the CAN Driver or the higher layer. -

		- - - - - The error will be raised if an incorrect PduId is calculated during reception in the function: -
13	`CANIF_E_PARAM_HRH`	The error code is used in the function - If an invalid hardware receive handle is passed to the CAN Interface.
14	`CANIF_E_PARAM_CHANNEL`	Not used.
15	`CANIF_E_PARAM_CONTROLLER`	Used by the following functions if an invalid controller index is passed: - - - - - - - - - The following function raises the error if an invalid controller index is calculated from a wake up source: - -
20	`CANIF_E_PARAM_POINTER`	The error is raised if a NULL pointer is passed to one of the following functions: - - - - - - -
30	`CANIF_E_UNINIT`	The error is raised if one of the following API functions is called before the CAN Interface is

		initialized: - - - - - - - - - -
40	`CANIF_E_NOK_NOSUPPORT`	Not used.
50	`CANIF_TRCV_E_TRANSCEIVER`	This error code notifies about an invalid transceiver index which is passed to one of the following functions: - - - - - -
60	`CANIF_TRCV_E_TRCV_NOT_STANDBY`	Not used.
70	`CANIF_TRCV_E_TRCV_NOT_NORMAL`	Not used.
80	`CANIF_E_INVALID_TXPDUID`	Not used (see `CANIF_E_PARAM_LPDU)`
90	`CANIF_E_INVALID_RXPDUID`	Not used (see `CANIF_E_PARAM_LPDU)`
Additionally defined error codes (not AUTOSAR compliant)		
45	`CANIF_E_CONFIG`	The error code CANIF_E_CONFIG is used to detect inconsistent data in the generated files due to misconfiguration. The error can be raised in the following functions: - -
46	`CANIF_E_FATAL`	The error code CANIF_E_FATAL is used to detect fatal errors. The DET error will be raises inside the queue handling if transmission won't be possible any more and if a not existent Pdu mode is requested in the function

		-

Table 3-3 Errors reported to DET

If the development error detection is disabled not only the reporting of the errors is suppressed but also the detection i.e. the verification of valid function parameters.

Production errors are reported to the Diagnostics Event Manger. These errors allow the ECU to continue operation. The following DEM errors are specified for the CAN Interface:

```
CANIF_E_STOPPED
CANIF_E_FULL_TX_BUFFER
CANIF_E_INVALID_DLC
```

The IDs of the error codes are assigned by the DEM at configuration time.

CANIF_E_STOPPED is reported during CanIf_Transmit if the controller does not reside in the mode CANIF_CS_STARTED.

CANIF_E_FULL_TX_BUFFER is reported if CanIf_Transmit is called for a PduId which is still buffered inside the CAN Interface. In this case the data of the queued message is overwritten and the DEM is informed. If no transmit buffer is active the DEM error will be raised if the Can_Write call returns CAN_BUSY due to an occupied hardware object.

CANIF_E_INVALID_DLC is raised during reception if the DLC of the received message is smaller than the DLC specified in the database (if the DLC check is activated).

The CAN Interface provides API and call back functions to control as many transceivers as CAN controllers are available in the system. The transceiver handling has to be activated at pre-compile time.

The CAN Interface provides the following functions for higher layers to control the behavior of the transceiver.

- CanIf_SetTransceiverMode()
- CanIf_GetTransceiverMode()
- CanIf_GetTrcvWakeupReason()
- CanIf_SetTransceiverWakeupMode()

The initialization of the transceiver driver itself is not executed by the CAN Interface. This means the calling layer has to make sure the transceiver driver in initialized before using the listed API functions.

If more than one different transceiver is used in the system the CAN Interface provides a mapping to address the correct transceiver driver with the correct parameters. The feature “Transceiver Mapping” has to be activated to control more than one transceiver driver.

It is also allowed to activate the feature “Transceiver Mapping” if only one transceiver driver is used in the system. But due to additional runtime it is suggested to deactivate this feature for this use case.

The CAN Interface supports the detection of wake up events raised by a transceiver. The feature “Sleep/WakeUp API” has to be activated and a “Wakeup source” has to be configured on the channel properties page in the CAN Interface for the channel which refers to the CAN Controller physically connected to the transceiver of interest.

The API CanIf_CheckWakeup() will request if a wake up event occurred on the transceiver addressed by the wakeup source passed in the parameter list of the call.

The CAN Interface analyses the passed wakeup source parameter and decides if a CAN Controller or a CAN Transceiver has to be asked for a pending wake up event.

For more details refer to the chapter 3.14 Sleep / WakeUp.

The CAN Interface controls the modes of the underlying CAN driver and transceiver driver. This means an API is provided to change the modes of the connected CAN transceivers and send a CAN controller to sleep.

The API function CanIf_SetControllerMode has to be used to change the mode of the CAN controller while the CAN transceiver has to be controlled with the API function CanIf_SetTransceiverMode.

⚠	The CAN Interface itself does not execute any checks if the CAN Controller and CAN Transceiver are set to sleep consistently and in the correct sequence. It is up to the higher layer to call CanIf_SetControllerMode() and CanIf_SetTransceiverMode() in the correct sequence.

Wake up events can be either raised by the CAN Controller or by the CAN Transceiver. In both cases the CAN Interface is not directly informed about state changes. This means the higher layers (normally the EcuM) has to call the CAN Interfaces CanIf_CheckWakeup API function with the wakeup sources configured for CAN Transceiver or CAN Interface (1).

The CAN Interface decides by analyzing the passed wakeup source if the CAN controller or a CAN Transceiver driver has to be checked for a pending wakeup (2 or 2'). If the requested layer returns a pending interrupt the EcuManager is notified by the call back function EcuM_SetWakeupEvent (3).

The following figure illustrates the described wake up sequence:

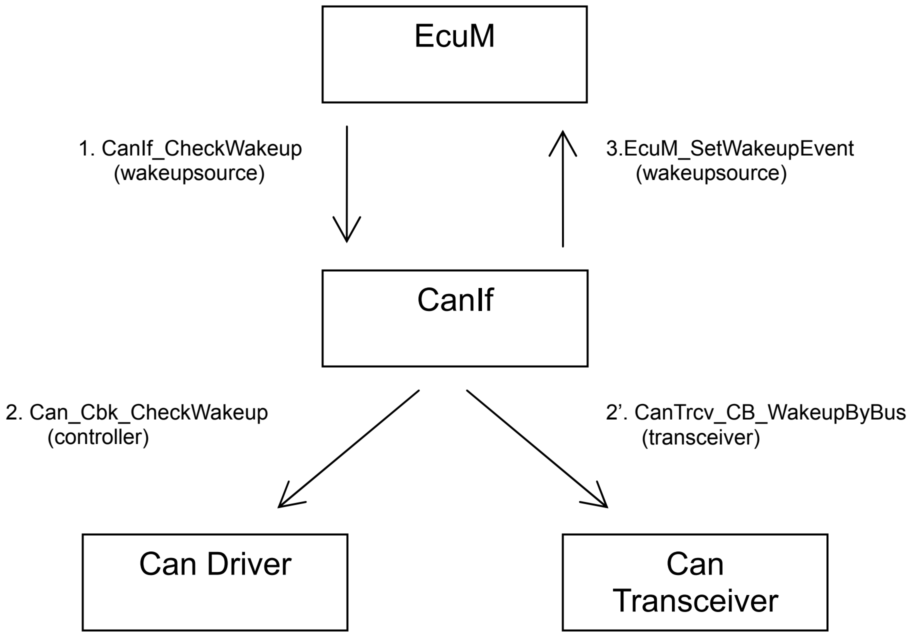

EcuM
1. CanIf_CheckWakeup (wakeupsource)
3.EcuM_SetWakeupEvent (wakeupsource)
CanIf
2. Can_Cbk_CheckWakeup (controller)
2'. CanTrcv_CB_WakeupByBus (transceiver)
Can Driver
Can Transceiver

Figure 3-1 Wake up sequence (No validation)

If the feature "Wake up validation" is activated the following figure shows the sequence which has to be executed for a valid wake up. Steps 1 to 3 take place as described above.

After the EcuM_SetWakeupEvent() call the CAN Interface has to be set to the state CANIF_CS_STARTED to be able to receive messages. These messages won't be passed to upper layers by the CAN Interface as the PDUMode is still set to "Offline". The state change which sets the CAN Interface to the started mode has to be realized by a call of the API function CanIf_SetControllerMode(controller, CANIF_CS_STARTED) (5) from the function EcuM_StartWakeupSources() (4). If the wake up was detected by the transceiver the CAN Controller has to be woken up internally. This means the call CanIf_SetControllerMode(controller, CANIF_CS_STOPPED) is necessary in (5) before the transition to started mode is executed.

If the wake up is initiated by the CAN controller, the transceiver has to be set to started mode and the CAN Controller has to be set to started mode.

If the wake up is initiated by the transceiver the CAN Controller has to be woken up internally. This means an additional call to CanIf_SetControllerMode(controller, CANIF_CS_STOPPED) has to be executed to wake up the CAN Controller before the transition to STARTED mode is initiated. (Depending on the behavior of the transceiver, the CAN Controller and the configuration there is the possibility to wake up both the CAN Controller and the Transceiver externally.)

Next the EcuManager starts a time out for the wake up validation. This means if a message is received within this timeout (6) the CanIf_CheckValidation() call executed by the EcuManager (7) will result in a successful validation. The CAN Interface checks for a recent Rx event (6) which occurred after the wake up and notifies the EcuManager by calling EcuM_ValidationWakeupEvent().

If there is no message reception after (5) the function CanIf_CheckValidation() calls won't notify a successful wake up validation and the EcuManager will run into a timeout. In this case the EcuManager calls EcuM_StopWakeupSources() (8') and the CAN Driver and CAN Transceiver have to be set to sleep mode again.

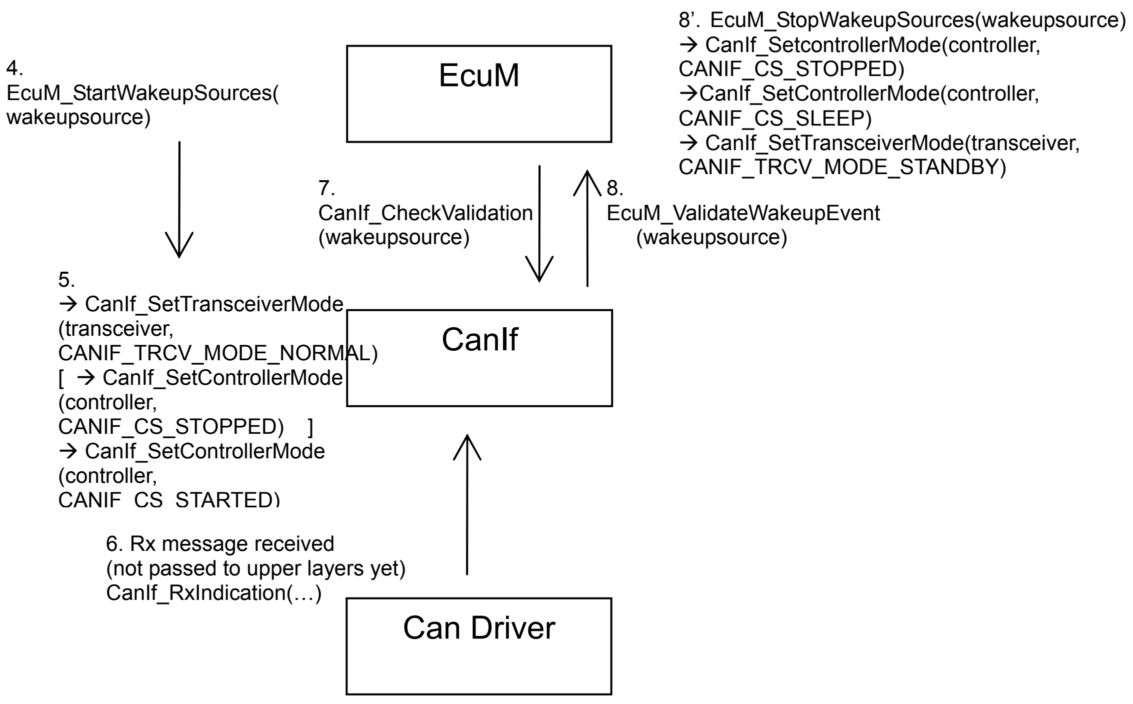

8'. EcuM_StopWakeupSources(wakeupsource)
→ CanIf_SetcontrollerMode(controller, CANIF_CS_STOPPED)
→CanIf_SetControllerMode(controller, CANIF_CS_SLEEP)
→ CanIf_SetTransceiverMode(transceiver, CANIF_TRCV_MODE_STANDBY)
4. EcuM_StartWakeupSources(wakeupsource)
EcuM
7. CanIf_CheckValidation (wakeupsource)
8. EcuM_ValidateWakeupEvent (wakeupsource)
5. → CanIf_SetTransceiverMode (transceiver, CANIF_TRCV_MODE_NORMAL)
[→ CanIf_SetControllerMode (controller, CANIF_CS_STOPPED)]
→ CanIf_SetControllerMode (controller, CANIF_CS_STARTED)
CanIf
6. Rx message received (not passed to upper layers yet) CanIf_RxIndication(…)
Can Driver

Figure 3-2 Wake up sequence (Wakeup validation)

During the wake up sequence as well as during the transition to sleep mode, the higher layers have to take care about the sequence of the state transitions affecting the CAN Controller (CAN Driver) and the Transceiver Driver.

The CAN Interface handles bus off events notified by the CAN Driver in interrupt driven or polling systems. If a bus off event is raised the CAN Driver forwards it to the CAN Interface by calling the function CanIf_ControllerBusOff.

The CAN Interface switches its internal controller state from STARTED to STOPPED and the PDU mode is set to OFFLINE.

In this state no reception or transmission is possible until the CAN Interface's controller state and as a result the CAN Controller's bus off state is recovered by a call to the function CanIf_SetControllerMode for the affected channel by the higher layer.

After the controller state is switched the bus off state is recovered. For successful reception and transmission the PDU mode has to be switched to RX_ONLINE, TX_ONLINE or ONLINE by the higher layer.

The version of the CAN Interface module can be acquired in three different ways. The first possibility is by calling the function CanIf_GetVersionInfo. This function will return the module's version in the structure Std_VersionInfoType which additionally includes the VendorID and the ModuleID.

The second possibility is the access of version defines which are specified in the header file CanIf.h.

The following defines can be evaluated to access different versions:

```
AUTOSAR version:
  CANIF_AR_MAJOR_VERSION
  CANIF_AR_MINOR_VERSION
  CANIF_AR_PATCH_VERSION
Module version:
  CANIF_SW_MAJOR_VERSION
  CANIF_SW_MINOR_VERSION
  CANIF_SW_PATCH_VERSION
Module ID:
  CANIF_MODULE_ID
Vendor ID:
  CANIF_VENDOR_ID
```

There is a third possibility to at least acquire the SW version by accessing globally visible constants:

CanIf_MainVersion, CanIf_SubVersion, CanIf_ReleaseVersion

The API function CanIf_GetVersionInfo is only available if enabled at compile time. The definitions can be accessed independent of the configuration.

In the following table services provided by other components which are used by the CAN Interface are listed. For details about prototype and functionality refer to the documentation of the providing component.

Component	***API***
DET	`Det_ReportError`
DEM	`Dem_ReportErrorStatus`

CAN Driver	`Can_InitController`
	`Can_SetControllerMode`
	`Can_Write`

Application (PDU Router, Network management, TP)	`User_TxConfirmation (*)`
	`User_RxIndication (*)`

Com Manager, Ecu Manager	`User_ControllerBusOff (*)`
	`User_SetWakeupEvent (*)`
	`User_ValidationWakeupEvent (*)`

Interrupt locks	`SchM_Enter_CanIf`
	`SchM_Exit_CanIf`
	`or`
	`VStdNestedGlobalInterruptDisable`
	`VStdNestedGlobalInterruptEnable`

Transceiver Driver	`CanTrcv_SetOpMode`
	`CanTrcv_GetOpMode`
	`CanTrcv_GetBusWuReason`
	`CanTrcv_SetWakeupMode`
	`CanTrcv_CB_WakeupByBus`

MICROSAR Extension (optional)	`EcuM_GeneratorCompatibilityError`

Table 3-4 API functions used by the CAN Interface

(*) names of the call back functions can be configured freely.

CanInterface must not be called during the corresponding area is entered. The CanInterface API `CanIf_CancelTxNotification()`/ `CanIf_CancelTxConfirmation()` mostely is entered via the CAN interrupt. For platforms for what the confirmation for a transmit cancelation needs to be polled, the corresponding API (for example `Can_MainFunction_Write()`) must not be called if the corresponding lock area is entered.

	CAN_EXCLUSIVE_AREA_0	CAN_EXCLUSIVE_AREA_1	CAN_EXCLUSIVE_AREA_2	CAN_EXCLUSIVE_AREA_3	CAN_EXCLUSIVE_AREA_4
CanIf_Init	■				■
CanIf_InitMemory	■				
CanIf_CheckWakeup		■	■	■	■
CanIf_Transmit		■	■	■	■
CanIf_CancelTransmit		■	■	■	■
CanIf_SetControllerMode			■	■	■
CanIf_ResetBusOffStart	■		■	■	■
CanIf_CancelTxNotificatio/ CanIf_CancelTxConfirmation		■	■	■	■
CanIf_SetPduMode					■

Table 3-6 Restrictions for the different lock areas

The CAN Interface implemented according to the AUTOSAR 3.0 specification is able to be used with a Communication Manager implemented according to AUTOSAR 2.1.

This compatibility mode implies changes in the API of the CAN Interface as well as changes in the call back functions to other layers. Refer to the next subchapters for details.

The feature AUTOSAR 2.1 ComM compliance does not implement any changes in the sleep/wake up behavior of the CAN Interface and therefore the following modules also have to be able to support a sleep/wake up behavior as specified in AUTOSAR 2.1:

- CAN Transceiver Driver / Input Capture Unit (ICU)
- CAN Driver
- ECU Manager (EcuM)

The compliance mode of the CAN Interface has to be configured at compile time. Refer to chapter 5.1 Module properties.

This chapter describes the API functions if the feature “AUTOSAR 2.1 ComM compliance” is activated. This description overrides the API description for the related functions in chapter 6 API Description. The changes in the API functions are highlighted red.

```
void CanIf_InitController(uint8 CanNetwork, CanIf_ControllerConfigType*
ConfigPtr);
```

```
Std_ReturnType CanIf_SetControllerMode(uint8 CanNetwork,
CanIf_ControllerModeType ControllerMode);
```

```
Std_ReturnType CanIf_GetControllerMode(uint8 CanNetwork,
CanIf_ControllerModeType  *ControllerModePtr)
```

```
Std_ReturnType CanIf_SetChannelMode(uint8 Channel, CanIf_
CanIf_ChannelSetModeType ChannelRequest)
```

replaces `CanIf_SetPduMode`

```
Std_ReturnType CanIf_GetChannelMode(uint8 Channel, CanIf_ChannelGetModeType *
ChannelModePtr)
```

replaces `CanIf_GetPduMode`

```
StdReturnType CanIf_SetTransceiverMode(uint8 CanNetwork,
CanIf_TransceiverModeType TransceiverMode)
```

```
StdReturnType CanIf_GetTransceiverMode(uint8 CanNetwork,
CanIf_TransceiverModeType *TransceiverModePtr)
```

```
StdReturnType CanIf_GetTrcvWakeupReason(uint8 CanNetwork,
CanIf_TrcvWakeupReasonType *TrcvWuReasonPtr)
```

```
StdReturnType CanIf_GetTrcvWakeupReason(uint8 CanNetwork,
CanIf_TrcvWakeupModeType TrcvWakeupMode)
```

The call back function used to notify a bus off to the CAN State Manager in AUTOSAR 3.0 is changed to notify the ComM. The prototype is defined as follows:

```
void CanSM_ControllerBusOff (ComM_ChannelHandleType CanNetwork)
```

According to the AUTOSAR 2.1 specifications the CAN Interface is responsible for the initialization of the underlying CAN Driver and CAN Transceiver Driver. This means the functions Can_Init() and CanTrcv_Init() are called from the CAN Interface and do not have to be called from any other layer.

```
CanIf_InitMemory();/* Optional call which reinitializes global
                   variables to set the CAN Interface back to
                   uninitialized state. */

/* Optional calls to initialize global variables */

CanTrcv_xxx_InitMemory();

Can_InitMemory();


CanIf_Init(<PtrToCanIfConfiguration>);

                   /* Global initialization of the CAN Interface,
                   all available controllers and then CAN Driver
                   are initialized within this call. If selectable
                   post build configuration is active a valid
                   configuration has to be passed to the call of
                   CanIf_Init. In other cases the parameter will be
                   ignored and a NULL pointer can be used */

CanIf_SetControllerMode(0, CANIF_CS_STARTED);

                   /* The controller mode for controller 0 is set
                   to started mode. This means the CAN controller
                   is initialized and ready to communicate
                   (acknowledge of the CAN controller is
                   activated). Communication is not yet possible
                   because the CAN Interface will neither pass Tx
                   PDUs from higher layers to the CAN Driver nor
                   accept Rx PDUs from the CAN Driver. */

CanIf_SetChannelMode(0, CANIF_SET_ONLINE);

                   /* The PDU mode in the CAN Interface is switched
                   to online mode. After initialization this mode
                   remains in the state CANIF_GET_OFFLINE until the
                   CanIf_SetPduMode function is called. Now
                   transmission requests will be passed from the
                   upper layer to the CAN Driver and Rx PDUs are
                   forwarded from the CAN Driver to the
                   corresponding higher layer. */
```

This chapter gives necessary information for the integration of the MICROSAR Can Interface into an application environment of an ECU.

The CAN Interface consists of the following files:

The delivery of the Can Interface contains the files which are described in the chapters 4.1.1 and 4.1.2:

CanIf.c	Implementation
CanIf.h	Header file; has to be included by higher layers to access the API
CanIf_cbk.h	Header file; has to be included by underlying layers to access call back functions provided by the CAN Interface
CanIf_Types.h	Definition of types provided by the CAN Interface which have to be used by other layers. This file will be automatically included if CanIf.h or CanIf_cbk.h is included.

Table 4-1 Static files

The dynamic files are generated by the configuration tool [config tool].

CanIf_cfg.h	Generated header file (included automatically by CanIf.h and CanIf_cbk.h)
CanIf_Lcfg.c	Contains link time configuration data. Contains data in case of Pre-compile, Link time and post build configuration variant.
CanIf_Pbcfg.c	Contains post build configuration data. If the “Link time configuration” variant is used, this file is empty.
CanIf_CanTrcv.h	Generated header file which includes the necessary header files of the transceivers used in the system.

Table 4-2 Generated files

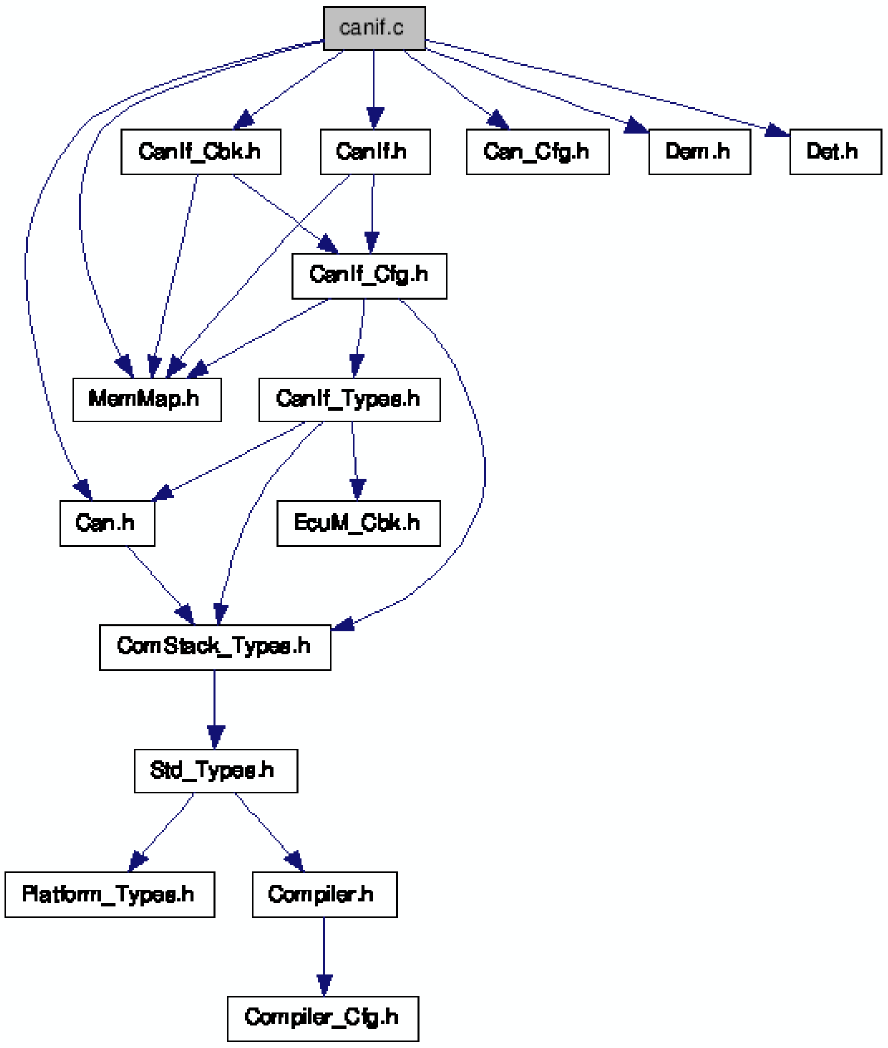


Figure 4-1 Include structure

The objects (e.g. variables, functions, constants) are declared by compiler independent definitions – the compiler abstraction definitions. Each compiler abstraction definition is assigned to a memory section.

The following table contains the memory section names and the compiler abstraction definitions defined for the CAN Interface and illustrates their assignment among each other.

	CANIF_VAR_ZEROINIT	CANIF_VAR_INIT	CANIF_VAR_NOINIT	CANIF_CONST	CANIF_PBCFG	CANIF_CODE	CANIF_APPL_CODE	CANIF_APPL_VAR	CANIF_APPL_PBCFG
CANIF_START_SEC_CODE CANIF_STOP_SEC_CODE						■			
CANIF_START_SEC_PBCFG CANIF_STOP_SEC_PBCFG					■				
CANIF_START_SEC_CONST_8BIT CANIF_STOP_SEC_CONST_8BIT				■					
CANIF_START_SEC_CONST_32BIT CANIF_STOP_SEC_CONST_32BIT				■					
CANIF_START_SEC_CONST_UNSPECIFIED CANIF_STOP_SEC_CONST_UNSPECIFIED				■					
CANIF_START_SEC_VAR_NOINIT_UNSPECIFIED CANIF_STOP_SEC_VAR_NOINIT_UNSPECIFIED			■						
CANIF_START_SEC_VAR_ZERO_INIT_UNSPECIFIED CANIF_STOP_SEC_VAR_ZERO_INIT_UNSPECIFIED	■								
CANIF_START_SEC_VAR_INIT_UNSPECIFIED CANIF_STOP_SEC_VAR_INIT_UNSPECIFIED		■							

Table 4-3 Compiler abstraction and memory mapping

The Compiler Abstraction Definitions CANIF_APPL_CODE, CANIF_APPL_VAR and CANIF_APPL_PBCFG are used to address code, variables and constants which are declared by other modules and used by the CAN Interface.

These definitions are not mapped by the CAN Interface but by the memory mapping realized in the CAN Driver, CAN Transceiver Driver, PDU Router, Network management, Transport Protocol Layer, ECU State Manager and the CAN State manager.

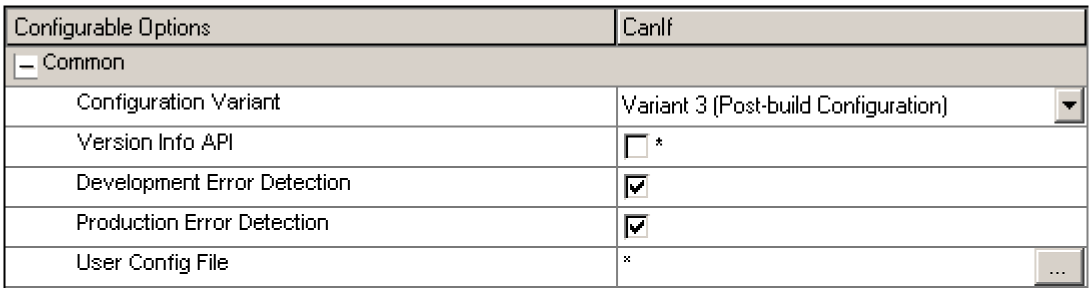
The CAN Interface supports pre-compile, link time and post build configuration.
In case of a library delivery the features listed in table 5-1 Pre-compile configuration have to be activated or deactivated before building the library.

The following chapters describe the configuration properties and the user interface of the configuration and generation tool GENy.

To enable the CAN Interface a channel has to be set up and a CAN Driver has to be added in the bottom left window.

The screenshot shows all available channel independent options for the CAN Interface.

Depending on the configuration variant some of the options won't be available. Some options will be preconfigured and cannot be changed by the user due to the features supported by other modules (e.g. CAN Driver)



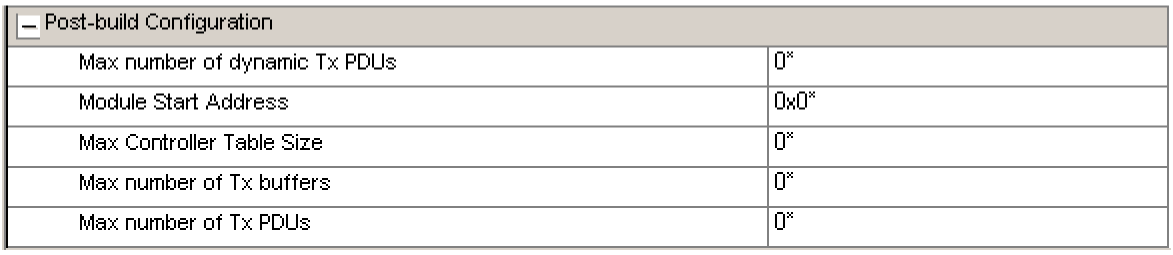
Configurable Options	CanIf
– Common	
Configuration Variant	Variant 3 (Post-build Configuration)
Version Info API	☐ *
Development Error Detection	✓
Production Error Detection	✓
User Config File	*

Figure 5-1 Module configuration (Common parameters, Pre-compile time)

Configuration Variant	Shows the currently active configuration class. This option will not be available to the user. Depending on the chosen configuration class some of the following option will not be available.
Version Info API	Enable the API function CanIf_GetVersionInfo
Development Error Detection	Enable the detection and reporting of development errors. (DET has to be available in the system)
Production Error Detection	Enables the detection and reporting of production errors. (DEM has to be available in the system)

User config file	Provides the possibility to concatenate the contents of the specified file to the generated file CanIf_Cfg.h

Table 5-1 Common configuration

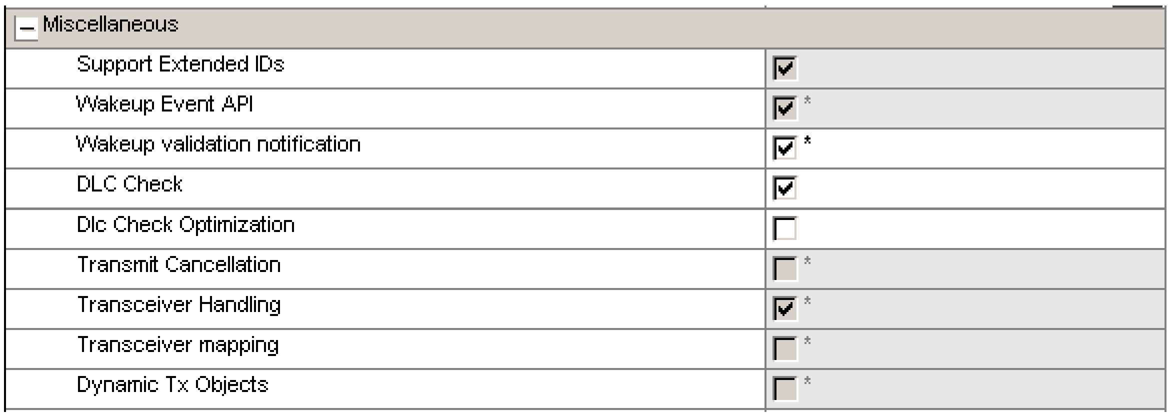


Post-build Configuration	
Max number of dynamic Tx PDUs	0*
Module Start Address	0x0*
Max Controller Table Size	0*
Max number of Tx buffers	0*
Max number of Tx PDUs	0*

Figure 5-2 Module configuration (Post build parameters pre-compile time)

Max number of dynamic Tx PDUs	Specify the maximum number of dynamic Tx PDUs in the system. The chosen value must be greater or equal to the number of dynamic Tx PDUs used during Postbuild phase.
Module Start Address	Specify the start address in the controller's ROM for the generated data of the CanInterface. (used for post-build configuration only)
Max Controller Table Size	Specify the maximum number of controllers used in the system. The chosen value must be greater or equal to the number of currently used controllers. For post build configurations the number of controllers is not allowed to exceed this value.(used for post-build configuration only)
Max number of Tx buffers	Specifies the maximum number of Tx PDUs which are buffered into the Tx queue. If the transmit buffer is activated, please enter the maximimum number of expected standard Tx object PDUs (PDUs which are not located into FullCAN).
Max number of Tx PDUs	Specifies the maximum number of Tx PDUs of the system. This declaration is only necessary if the "Bit queue" is active. If the transmit buffer of type "Bit queue" is actived, please enter the maximum number of expected Tx object PDUs (Standard and FullCAN Tx Pdus inclusive dynamic Tx PDUs).

Table 5-2 Post build configuration



Miscellaneous	
Support Extended IDs	✓
Wakeup Event API	✓ *
Wakeup validation notification	✓ *
DLC Check	✓
Dlc Check Optimization	☐
Transmit Cancellation	☐ *
Transceiver Handling	✓ *
Transceiver mapping	☐ *
Dynamic Tx Objects	☐ *

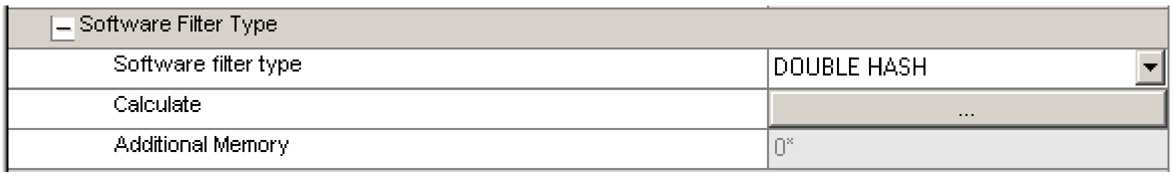
Figure 5-3 Module configuration (Miscellaneous, Pre-compile time)

Support extended IDs	Enable this option if extended or mixed identifiers have to be handled by the CAN Interface.
Wakeup Event API	Enable/Disable wake up support. If this option is activated the CanIf_CheckWakeup function will identify wake up events raised by either the CAN Driver or the Transceiver Driver. If this feature is deactivated the CanIf_CheckWakeup function will always return E_NOT_OK.
Wakeup Validation Notification	Enable wake up validation support. The function CanIf_CheckValidation will be available to check for a validation of a wake up event. If this feature is deactivated the function will always return E_NOT_OK.
DLC check	Enables the DLC check of Rx PDUs while reception.

Dlc Check Optimization	Use this parameter to choose between DLC check against optimized DLC and DLC check against not optimized DLC. Optimized DLC means that only data bytes of a CAN message which are allocated by defined signals within your data base are taken into account of calculation of DLC which is used for DLC check. Not optimized DLC means that the specified DLC of a CAN message within you data base is used for DLC check independent of allocation of data bytes by signals. If enabled DLC check is performed against optimized DLC. If disabled DLC check is performed against not optimized DLC. RESTRICTION: This parameter is only editable if 'DLC Check' is enabled.
Transmit cancellation	Enables the cancellation of PDUs which are already passed to the CAN Driver. If a message with higher priority will be transmitted the previous message will be cancelled and re-queued in the CAN Interface (Depends on the used hardware if this feature is supported)
Transceiver handling	Enable/Disable the API and call back functions to interact with a transceiver driver as specified for AUTOSAR. The following functions will be available if transceiver handling is activated: - CanIf_SetTransceiverMode() - CanIf_GetTransceiverMode() - CanIf_GetTrcvWakeupReason() - CanIf_SetTransceiverWakeupMode() and the identification of wake up events raised by the transceiver using the CanIf_CheckWakeup() API.

Transceiver mapping	If this feature is enabled the CAN Interface support multiple different transceiver drivers. This means the API functions - CanIf_SetTransceiverMode() - CanIf_GetTransceiverMode() - CanIf_GetTrcvWakeupReason() - CanIf_SetTransceiverWakeupMode() convert the passed transceiver index to the correct Transceiver Driver instance. This feature can also be activated if only one transceiver driver is used in the systems (for optimized runtime it is suggested to disable this feature if only one transceiver driver is used in the system)
Dynamic Tx Objects	If this feature is enabled the CAN Interface supports dynamic Tx Objects. For details please look for chapter “Dynamic transmission”.

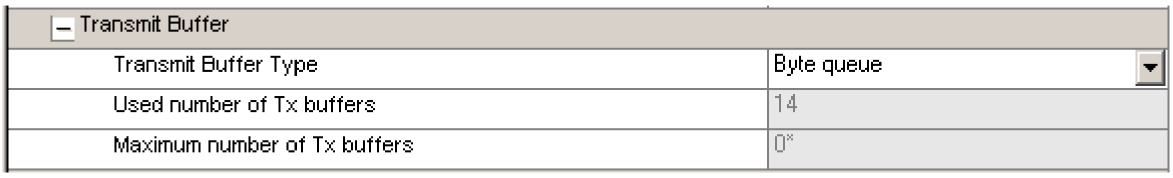
Table 5-3 Miscellaneous configuration



Software Filter Type	
Software filter type	DOUBLE HASH
Calculate	...
Additional Memory	0*

Figure 5-4 Software filter type configuration (Pre-compile time)

Software Filter Type	Choose the search algorithm for the software filtering. <u>Linear Search</u>: very efficient for a small amount of Rx PDUs. Search time increases linearly for Rx PDUs with low priority <u>Binary Search</u>: nearly constant search time for all PDUs. More efficient for big amounts of Rx PDUs <u>Double Hash</u>: always constant search time for all PDUs
Calculate	Press this button to get better results for the double hash algorithm, i.e. the amount of memory could be reduced by pressing this button repeatedly.
Additional Memory	Additional memory compared to linear search algorithm. It's only an estimation.

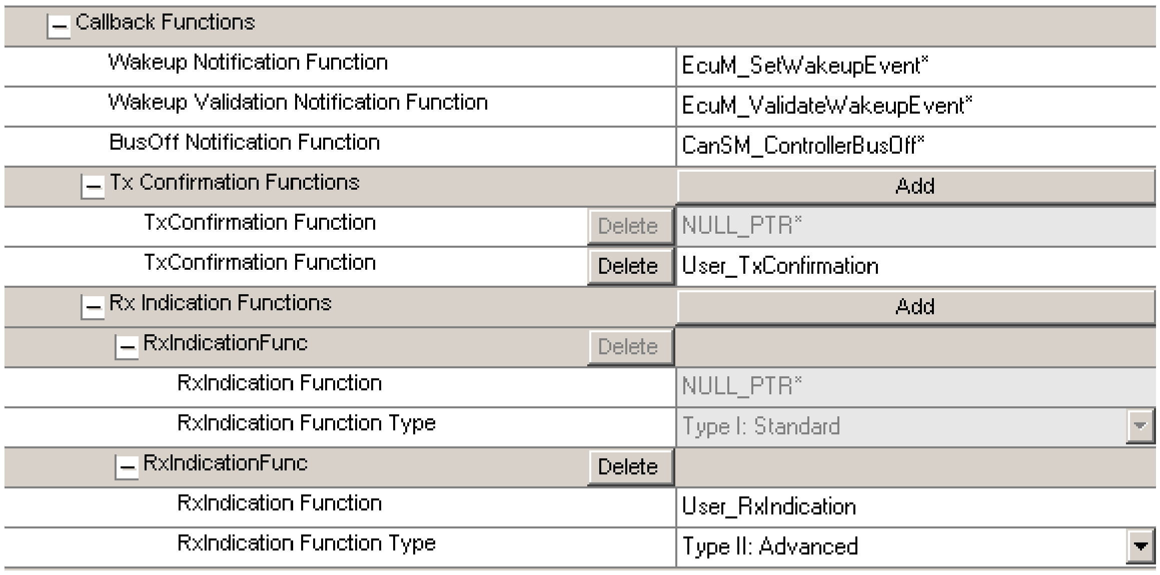


Transmit Buffer	
Transmit Buffer Type	Byte queue
Used number of Tx buffers	14
Maximum number of Tx buffers	0*

Figure 5-5 Transmit buffer configuration (Pre-compile time)

Transmit Buffer	Selects the type of the Tx transmit buffer. Basic CAN Tx PDUs are buffered in the CAN Interface, if the hardware is busy while the function CanIf_Transmit is called. The type “Byte queue” consumes more RAM as the type “Bit queue” but is a little bit faster.
Used number of transmit buffers (read only)	Shows the amount of used elements of the transmit buffer. In case of pre-compile and link-time configuration the size of the transmit buffer is equal to the size of the used elements.
Maximum number of the transmit buffers (read only)	Shows the size of the transmit buffer in case of post-build configurations. In post-build configurations the queue size has to be defined at link-time. For this reason elements can be reserved for changes in post-build configurations e.g. additional Tx messages / moving Tx messages from Tx FullCAN to Normal Tx. -> See section “Post build configuration” configuration item “Max Tx Pdu Handle Table Size”

Table 5-4 Transmit buffer configuration



Callback Functions		
Wakeup Notification Function		EcuM_SetWakeupEvent*
Wakeup Validation Notification Function		EcuM_ValidateWakeupEvent*
BusOff Notification Function		CanSM_ControllerBusOff*
Tx Confirmation Functions		Add
TxConfirmation Function	Delete	NULL_PTR*
TxConfirmation Function	Delete	User_TxConfirmation
Rx Indication Functions		Add
RxIndicationFunc	Delete	
RxIndication Function		NULL_PTR*
RxIndication Function Type		Type I: Standard
RxIndicationFunc	Delete	
RxIndication Function		User_RxIndication
RxIndication Function Type		Type II: Advanced

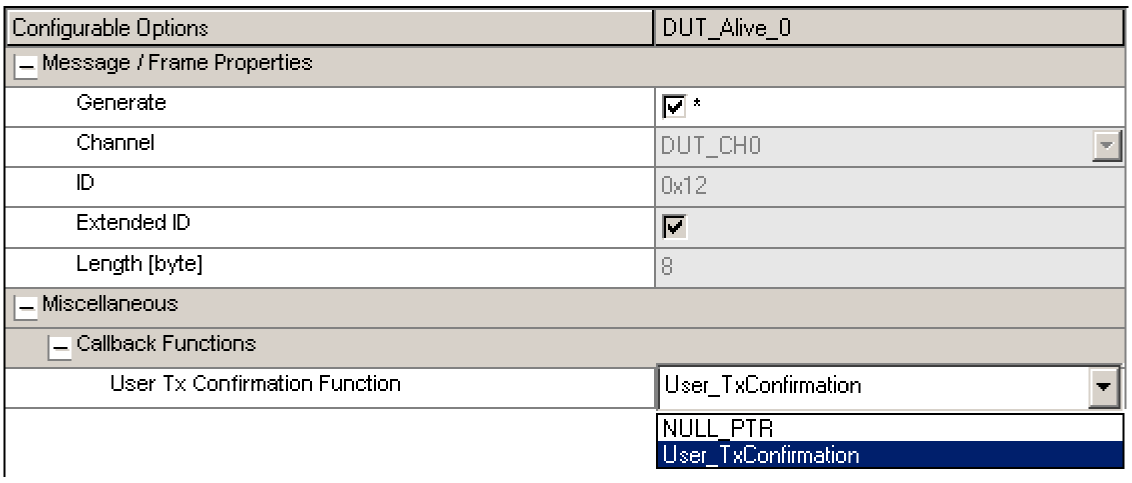
Figure 5-6 Call back function configuration (Link time)

Wakeup Notification Function	Specify the name of a wake up notification function. (e.g. EcuM_SetWakeupEvent)
Wakeup Validation Notification Function	Specify the name of the wake up validation function (e.g. EcuM_ValidateWakeupEvent)

NmOsek RxIndication support

EcuM Wakeup Source ID (Transceiver)	For each channel a wake up source for a connected transceiver can be specified. If this value is not equal to 0 the CAN Interface uses the specified wake up source to identify a wake up event raised by a transceiver. The value has to be chosen accordingly to the configuration of the Ecu Manager.
Range Configuration	A new range can be created by using the “Add” button. The range is defined by a value for code and a value for the mask. Messages which pass the range are calculated using the following formula: <Received ID> & <Mask> == <Code> For each range an RxIndication function has to be chosen. The function has to be created on the module page. If extended IDs have to pass the range the “Extended IDs” checkbox has to be activated. (Only visible if the “Extended ID support” on the module page is activated) For ranges a DLC check against a data length of 8 is executed. This mechanism can be avoided by checking the “Disable DLC check” feature.

Table 5-7 Channel specific properties

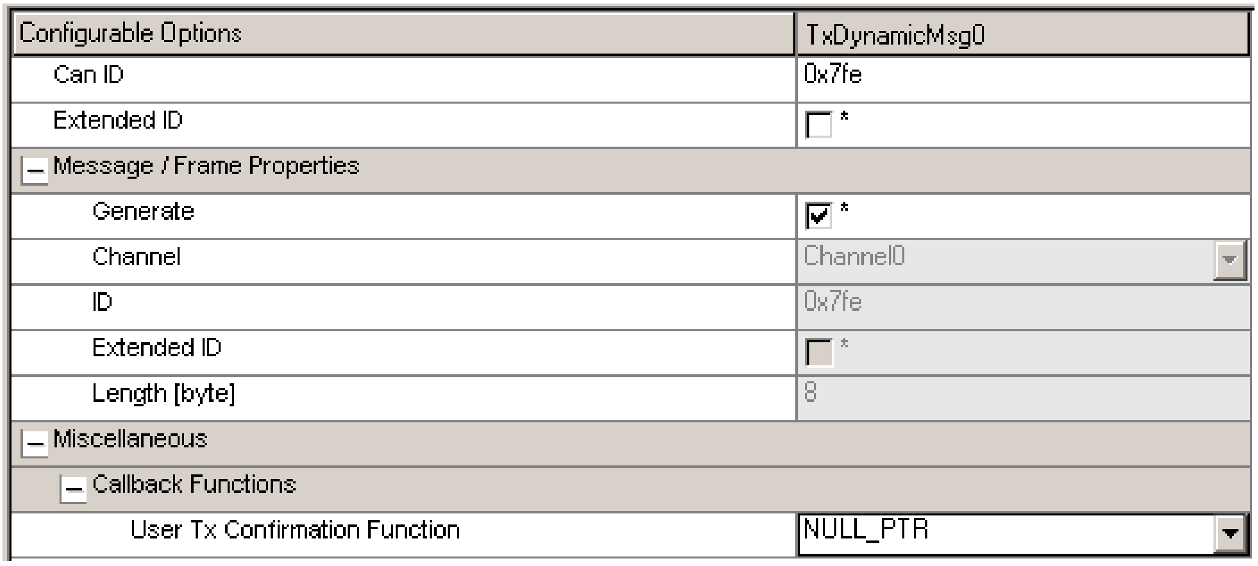


Configurable Options	DUT_Alive_0
Message / Frame Properties	
Generate	✓ *
Channel	DUT_CH0
ID	0x12
Extended ID	✓
Length [byte]	8
Miscellaneous	
Callback Functions	
User Tx Confirmation Function	User_TxConfirmation

Figure 5-9 Tx message properties

Generate	Activate if this message has to be enabled in the configuration. If this checkbox is disabled the message will not be known by the ECU.
User Tx Confirmation Function	Select a Tx Confirmation function from the list or the NULL_PTR if no confirmation function is desired. The name of the confirmation function has to be configured on the module page.

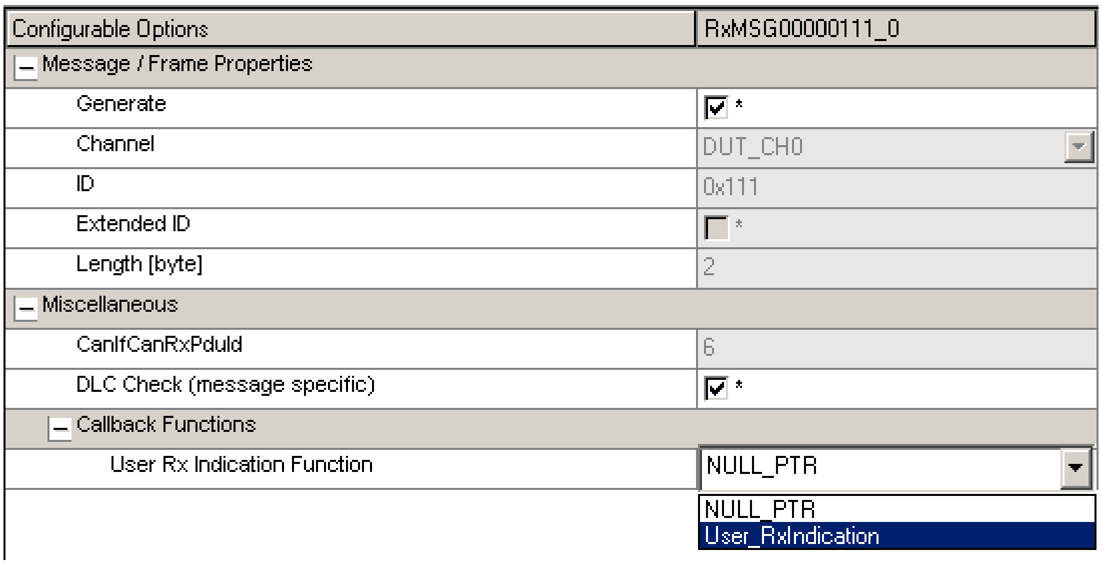
Table 5-8 Tx message properties



Configurable Options	TxDynamicMsg0
Can ID	0x7fe
Extended ID	☐ *
− Message / Frame Properties	
Generate	☑ *
Channel	Channel0
ID	0x7fe
Extended ID	☐ *
Length [byte]	8
− Miscellaneous	
− Callback Functions	
User Tx Confirmation Function	NULL_PTR

Table 5-9 Dynamic Tx message properties

Can ID	The CAN ID for the message which will be set after the initialisation of the CanInterface. This initial ID also defines the priority of the dynamic object.
Extended ID	If this flag is set, the inscribed CAN ID will be interpreted as extended ID. Never the less, during runtime the dynamic object can be overwritten with a standard CAN ID.



Configurable Options	RxMSG00000111_0
Message / Frame Properties	
Generate	✓ *
Channel	DUT_CH0
ID	0x111
Extended ID	*
Length [byte]	2
Miscellaneous	
CanIfCanRxPduId	6
DLC Check (message specific)	✓ *
Callback Functions	
User Rx Indication Function	NULL_PTR

Figure 5-10 Rx message properties page

Generate	Activate if this message has to be enabled in the configuration. If this checkbox is disabled the message will not be known by the ECU.
CanIfCanRxPduID (read only)	Shows the PDU ID for the currently selected message which is passed to the higher layers in the parameter list of the User_RxIndication function. (This value is valid after executing the generation process)
DLC Check (message specific)	If the “DLC check” is activated on the module page the DLC check can be avoided for single Rx messages by un-checking the “Message specific DLC Check” check box. If the “DLC check” is deactivated on the module configuration page this check box has no effect.
User Rx Indication Function	Select an Rx Indication function from the list or the NULL_PTR if no indication function is desired. The name and type of the indication function has to be configured on the module page.

Table 5-10 Rx message properties

`void CanIf_GetVersionInfo( Std_VersionInfoType *VersionInfo );`	
`Versioninfo`	Pointer to the structure including the version information.
-	-
Function to acquire version information	
The function is only available if enabled at compile time (CANIF_VERSION_INFO_API = STD_ON)	

Table 6-1 API CanIf_GetVersionInfo

`void CanIf_Init( const CanIf_ConfigType *ConfigPtr )`	
`CanIf_CtrlIdx`	The index to the Init structure used for initialization. (Not supported in current implementation)
`ConfigPtr`	Pointer to the structure including configuration data.
-	-
This function initializes global CAN Interface variables during ECU start-up and initiates the initialization of the CAN Controllers.	
Has to be called during start-up before CAN communication. Can_Init() has to be successfully executed.	

Table 6-2 API CanIf_Init

`void CanIf_InitController(uint8 Controller, uint8 ConfigurationIndex)`

`Controller`	The Controller to be initialized.
`ConfigurationIndex`	Not supported parameter
-	-
This function initializes the transmit buffer for the specified controller and calls the initialisation function for the corresponding controller of the CAN driver.	
Has to be called during start-up before CAN communication.	

Table 6-3 API CanIf_InitController

`Std_ReturnType` **`CanIf_SetControllerMode`**`(uint8 Controller, CanIf_ControllerModeType ControllerMode)`	
`Controller`	The Controller to change mode.
`ControllerMode`	Mode request.
Std_ReturnType	Returns whether the state transition was successful.
Request the mode of the specified channel. Supported modes: CANIF_CS_SLEEP, CANIF_CS_STOPPED, CANIF_CS_STARTED	
CAN Interface has to be initialized.	

Table 6-4 API CanIf_SetControllerMode

Prototype	
`Std_ReturnType` **`CanIf_GetControllerMode`**`(uint8 Controller, CanIf_ControllerModeType *ControllerModePtr)`	
`Controller`	Request mode of specified Controller.
`ControllerModePtr`	Pointer to data type the information is stored in.
Std_ReturnType	Returns whether the state request was successful.

Acquire the current controller mode of the specified channel
CAN Interface has to be initialized.

Table 6-5 API CanIf_GetControllerMode

`Std_ReturnType` **`CanIf_Transmit`**`(PduIdType CanTxPduId,` **`const`** `PduInfoType *PduInfoPtr)`	
`CanTxPduId`	Handle of the Tx PDU which will be transmitted.
`PduIndoPtr`	Pointer to a struct containing the properties of the Tx PDU.
Std_ReturnType	Returns if the transmit request was accepted.
Requests the transmission of the specified Tx PDU.	
- CAN Interface has to be initialized - Must not be called re-entrant	

Table 6-6 API CanIf_Transmit

`void CanIf_TxConfirmation``(PduIdType CanTxPduId)`	
`CanTxPduId`	ID of the successfully transmitted PDU.
-	-
Confirms the successful transmission of a Tx PDU	
CAN Interface has to be initialized.	

Table 6-7 API CanIf_TxConfirmation

`void CanIf_RxIndication``(uint8 Hrh, Can_IdType CanId, uint8 CanDlc,` **`const`** `uint8 *CanSduPtr)`

`Hrh`
`CanId`
`CanDlc`
`CanSduPtr`
	Hardware handle the PDU was received in.
	CAN identifier of the received PDU.
	Data length code of the received PDU.
	Pointer to hardware or temporary buffer containing the data of the received PDU.

-	-
The CAN Driver notifies the CAN Interface about a received Rx PDU.	
CAN Interface has to be initialized.	

Table 6-8 API CanIf_RxIndication

`void` **`CanIf_ControllerBusOff`**`(uint8 Controller)`	
`Controller`	Affected controller.
-	-
Indicates a BusOff for the specified controller to the CAN Interface.	
CAN Interface has to be initialized.	

Table 6-9 API CanIf_ControllerBusOff

`Std_ReturnType` **`CanIf_SetPduMode`**`(uint8 Controller, CanIf_PduSetModeType PduModeRequest)`	
`Controller` `PduModeRequest`	Controller which will be affected by the new Pdu mode. Requested Pdu mode
`Std_ReturnType`	Returns whether the state request was successful.

Change mode for specified controller. Possible states are:
`CANIF_SET_OFFLINE,`
`CANIF_SET_RX_OFFLINE,`
`CANIF_SET_RX_ONLINE,`
`CANIF_SET_TX_OFFLINE,`
`CANIF_SET_TX_ONLINE,`
`CANIF_SET_ONLINE,`
`CANIF_SET_TX_OFFLINE_ACTIVE`

CAN Interface has to be initialized. Controller has to be in state CANIF_CS_STARTED.

Table 6-10 API CanIf_SetPduMode

`Std_ReturnType` **`CanIf_GetPduMode`**`(uint8 Controller, CanIf_PduGetModeType * PduModePtr)`	
`Controller`	Request mode of the specified Controller.
`PduModePtr`	Pointer to a data buffer the current mode will be stored in.
`Std_ReturnType`	Returns whether the request of the current state was successful.
Request the current mode of the specified controller..	
CAN Interface has to be initialized.	

Table 6-11 API CanIf_GetPduMode

`void CanIf_InitMemory``(`**`void`**`)`	
-	-
-	-
Initializes global RAM variables, which have to available before any call to the CanIf API.	
May only be called once before CanIf_Init().	

Table 6-12 API CanIf_InitMemory

void CanIf_CancelTxconfirmation(const Can_PduType *PduInfoPtr)	
PduInfoPtr	Contains information about cancelled PDU
-	-
Called by the CAN Driver to notify the CAN Interface about a cancelled PDU which has to be re-queued.	
only available if CANIF_TRANSMIT_CANCELLATION = STD_ON is set.	

Table 6-13 API CanIf_CancelTxConfirmation

StdReturnType CanIf_SetTransceiverMode(uint8 Transceiver, CanIf_TransceiverModeType TransceiverMode)	
Transceiver	Address the transceiver by a transceiver index.
TransceiverMode	Requested mode transition
Std_ReturnType	Returns whether the state transition was successful.
Called by an upper layer to set the transceiver to another mode.	
Only available if transceiver handling is activated at configuration time. (CANIF_TRCV_HANDLING = STD_ON)	

Table 6-14 API CanIf_SetTransceiverMode

StdReturnType CanIf_GetTransceiverMode(uint8 Transceiver, CanIf_TransceiverModeType *TransceiverModePtr)	
Transceiver	Address the transceiver by a transceiver index.
TransceiverModePtr	Pointer to a buffer where current transceiver mode can be stored in.
Std_ReturnType	Returns whether the request of the current transceiver mode was successful.

Called by an upper layer to request the current mode of the transceiver.
Only available if transceiver handling is activated at configuration time. (CANIF_TRCV_HANDLING = STD_ON)

Table 6-15 API CanIf_GetTransceiverMode

StdReturnType CanIf_GetTrcvWakeupReason(uint8 Transceiver, CanIf_TrcvWakeupReasonType *TrcvWuReasonPtr)	
Transceiver	Address the transceiver by a transceiver index.
TrcvWuReasonPtr	Pointer to a buffer where the transceiver's wake up reason can be stored in.
Std_ReturnType	Returns whether the request of the wake up reason was successful.
Called by an upper layer to request the wake up reason stored in the transceiver.	
Only available if transceiver handling is activated at configuration time. (CANIF_TRCV_HANDLING = STD_ON)	

Table 6-16 API CanIf_GetTrcvWakeupReason

StdReturnType CanIf_GetTrcvWakeupReason(uint8 Transceiver, CanIf_TrcvWakeupModeType TrcvWakeupMode)	
Transceiver	Address the transceiver by a transceiver index.
TrcvWakeupModeType	Enable, disable or clear notification for wake up events.
Std_ReturnType	Returns whether the requested mode was set successfully.
Called by an upper layer to enable, disable or clear the wake up event notification of the transceiver.	
Only available if transceiver handling is activated at configuration time. (CANIF_TRCV_HANDLING = STD_ON)	

Table 6-17 API CanIf_SetTransceiverWakeupMode

Std_ReturnType CanIf_CheckWakeup(EcuM_WakeupSourceType WakeupSource)	
`WakeupSource`	Wakeup source which identifies the possible wakeup source (Transceiver / CAN Controller)
`Std_ReturnType`	Returns whether the received wakeup source was valid and the function could be executed correctly.
Called by an upper layer to check if a transceiver or CAN controller recently raised a wakeup. If a wakeup was detected from either transceiver or CAN Controller the EcuM call back function EcuM_SetWakeupEvent is called from the function's context.	
CAN Interface has to be initialized.	

Table 6-18 API CanIf_CheckWakeup

Std_ReturnType CanIf_CheckValidation(EcuM_WakeupSourceType WakeupSource)	
`WakeupSource`	Wakeup source which identifies the possible wakeup source (Transceiver / CAN Controller)
`Std_ReturnType`	Returns whether the requested mode was set successfully.
Called by an upper layer to check if a first Rx message was received after a wake up occurred from one of the supported sources. If a message was received between the call of CanIf_CheckWakeup and CanIf_CheckValidation the configurable EcuM call back function EcuM_ValidationWakeupEvent is called from the context of this function.	
CAN Interface has to be initialized. CanIf_CheckWakeup has to be called before and a wake up event has to be detected. CANInterface has to be set to CANIF_CS_STARTED mode before a validation is possible.	

Table 6-19 API CanIf_CheckValidation

void CanIf_ResetBusOffStart(uint8 Controller)

`Controller`	Recover bus off for the specified controller
-	-
Initiates the bus off recovery for a specified channel. A call to CanIf_ResetBusOffEnd has to follow on task level.	
Non-Autosar compliant API function which has to be enabled by defining CANIF_BUSOFF_RECOVERY_API = STD_ON	

Table 6-20 API CanIf_ResetBusOffStart

`void CanIf_ResetBusOffEnd``(uint8 Controller)`	
`Controller`	Recover bus off for the specified controller
-	-
Finishes the bus off recovery for a specified channel. A call to CanIf_ResetBusOffStart has to be executed before.	
Non-Autosar compliant API function which has to be enabled by defining CANIF_BUSOFF_RECOVERY_API = STD_ON The function has to be called on task level.	

Table 6-21 API CanIf_ResetBusOffEnd

`Std_ReturnType CanIf_ConvertPduId``(PduIdType PbPduId, PduIdType* PduId)`	
`PbPduId` `PduId`	Convert the Pdu specified by PbPduId Pointer to a buffer to store converted PduId.
`Std_ReturnType`	Returns whether the requested PduID was successfully converted.

Finishes the bus off recovery for a specified channel. A call to CanIf_ResetBusOffStart has to be executed before.
Non-Autosar compliant API function which cannot be activated by the user (CANIF_SUPPORT_NONPB_API = STD_ON)

Table 6-22 API CanIf_ConvertPduId

void CanIf_CancelTransmit (PduIdType CanTxPduId)	
`CanTxPduId`	PduId of the message which has to be cancelled
-	-
Initiate a cancellation / suppression of the confirmation of a Tx message.	
CAN Interface has to be initialized. Non-Autosar compliant API function which has to be enabled by defining CANIF_CANCEL_SUPPORT_API = STD_ON	

Table 6-23 API CanIf_CancelTransmit

void CanIf_CancelTxNotification (PduIdType PduId, CanIf_CancelResultType IsCancelled)	
`PduId` `IsCancelled`	Id of the Tx message which was cancelled Parameter currently not evaluated.
-	-

Called by the CAN Driver to notify about a cancelled message. No confirmation will be raised for this message.
CAN Interface has to be initialized. Non-Autosar compliant API function which has to be enabled by defining CANIF_CANCEL_SUPPORT_API = STD_ON

Table 6-24 API CanIf_CancelTxNotification

void CanIf_SetDynamicTxId(PduIdType CanTxPduId, Can_IdType CanId)	
CanTxPduId	PDU ID of the Tx message
CanId	CAN ID of the messageParameter
-	-
Called by the application to set the CAN Id of the corresponding Tx PDU.	
CAN Interface has to be initialized. Must not be interrupted by a call of CanIf_Transmit() for the same Tx PDU.	

Table 6-25 API CanIf_SetDynamicTxId

`void` **`EcuM_GeneratorCompatibilityError`**`(uint16 CanIfModuleId, uint8 CanIfInstanceId )`	
CanTpModuleId	Contains the CANIF_MODULE_ID (60) as defined by AUTOSAR.
CanTpInstanceId	For the CanIf only one instance is available, so this value is always zero.
None	
Called once by the CanIf during the initialization phase to indicate one of the following possible errors: - Abort initialization as generator is not compatible, - Abort initialization as current configuration is not compatible with the pre-compile configuration - Abort initialization as current configuration is not compatible with the link-time configuration	
None	
Call Context	
■ This function is either called in the CanIf_ChangeParameterRequest callers context or from the CanIf main function in task context.	

Table 6-26 EcuM_GeneratorCompatibilityError

Following restrictions apply to the current CAN_IF implementation.

The API CanIf_Transmit() must not be called re-entrant.

Dynamic transmit L-PDUs are buffered based on the initial assigned CAN ID and not on the CAN ID assigned by the API `CanIf_SetDynamicTxId()`.

EAD	Embedded Architecture Designer; generation tool for MICROSAR components
GENy	Generation tool for CANbedded and MICROSAR components

Table 8-1 Glossary

API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
CANSM	CAN State Manager
COMM	Communication Manager
DEM	Diagnostic Event Manager
DET	Development Error Tracer
DLC	Data Length Code
EAD	Embedded Architecture Designer
ECU	Electronic Control Unit
ECUM	ECU State Manager
HRH	Hardware Receive Handle
HTH	Hardware Transmit Handle
ISR	Interrupt Service Routine
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
PDU	Protocol Data Unit
SDU	Service Data Unit
SRS	Software Requirement Specification
SWC	Software Component
SWS	Software Specification

Table 8-2 Abbreviations

Visit our website for more information on

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