VECTOR

MICROSAR CAN Transceiver Driver

Technical Reference

Generic

Version 4.02.01

Authors	Matthias Fleischmann, Senol Cendere, Mihai Olariu, Timo Vanoni, Eugen Stripling, Jan Hammer
Status	Released

1 Document Information

1.1 History

Author	Date	Version	Remarks
Matthias Fleischmann	2008-04-21	1.00	Creation
Senol Cendere	2008-06-10	1.01	Rework file names Rework tool description
Mihai Olariu	2008-07-10	1.02	Rename the placeholders
Mihai Olariu	2008-10-13	1.03	Minor changes in the GENy GUI
Mihai Olariu	2008-12-15	1.04	Add support for the platforms which cannot wakeup by CAN bus activity
Matthias Fleischmann	2009-07-01	1.05	Updated description for ICU notification function and GENy configuration. Added description for BSWMD file configuration.
Timo Vanoni	2009-11-12	1.06	Added API CanTrcv_30_<Your_Trcv>_Wait() Filenames were renamed to match BSW00347. Change of initialization flow.
Timo Vanoni	2010-05-06	1.06.1	Fixed example in chapter 4.5
Timo Vanoni	2011-01-26	1.07.00	Added support for Identity Manager Configurations
Timo Vanoni	2011-07-19	2.00.00	Implementation according ASR3.2.1
Timo Vanoni	2011-12-06	2.01.00	Add support for selective wakeup (partial networking) Add support for SPI Interface
Timo Vanoni	2012-08-22	3.00.00	Add support for SetPNActivationState API (ch. 6.1.16) Change wakeup detection of CB_WakeupByBus to conform to ASR3.2.2 (ch.6.1.8)
Timo Vanoni	2012-09-28	3.01.00	Add support for ASR4.0.3
Timo Vanoni	2013-04-12	3.02.00	Small reworks, add note in chapter 4.4

Eugen Stripling	2014-10-20	4.00.00	AUTOSAR3 removed, Post-build selectable, access to generated data changed
Eugen Stripling	2015-06-19	4.01.00	Adapted according to ESCAN00082535
Eugen Stripling	2015-12-15	4.02.00	Chapter 8.2.7 added, chapter 3.2 adapted
Jan Hammer	2020-01-07	4.02.01	Update document format

Table 1 History of the document

1.2 Reference Documents

No.	Title	Version
[1]	AUTOSAR_SWS_CAN_TransceiverDriver.pdf	3.0.0
[2]	AUTOSAR_SWS_DET.pdf	2.2.1
[3]	AUTOSAR_SWS_DEM.pdf	2.2.0
[4]	AUTOSAR_BasicSoftwareModules.pdf	V1.0.0

Table 2 Reference documents

1.3 Scope of the Document

This technical reference describes the specific use of the Generic CAN transceiver driver. Note that the substrings “__Your_Trcv__” and “__YOUR_TRCV__” are just placeholders for the real name of the CAN transceiver (e.g. Tja1041 and TJA1041).

Please note
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.

Contents

Illustrations

Tables

2 Introduction

This document describes the functionality, API and configuration of the AUTOSAR BSW module CANTRCV as specified in [1]. The CAN transceiver driver provides an abstraction layer for the used CAN transceiver hardware. It offers a hardware independent interface to the upper layer components.

Supported AUTOSAR Release:	4	
Supported Configuration Variants:	Pre-compile, Post-build selectable	
Vendor ID:	CANTRCV_30___YOUR_TRCV___VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	CANTRCV_30___YOUR_TRCV___MODULE_ID	70 (according to ref. **[4]**)
Instance ID:	CANTRCV_30___YOUR_TRCV___INSTANCE_ID	Specified as pre-compile parameter in the Generation Tool.

Info
Replace the placeholders __Your_Trcv__ and __YOUR_TRCV__ with the according name of the used transceiver.

__YOUR_TRCV__ is used for definitions in upper case (e.g. TJA1041).

__Your_Trcv__ is used for variables in camel case (e.g. Tja1041).

In BSWMD file the placeholder is set per default to Generic. Please adapt this one according to the name of used transceiver as well.

3 Functional Description

3.1 Features

The features listed in this chapter cover the complete functionality specified in [1].

The supported and not supported features are presented in the following two tables.

Supported features
CAN Transceiver initialization.
CAN Transceiver control via DIO.
Detection of wakeup (optional: incl. partial networking)
Getting and setting operation mode of CAN transceiver.
CAN Transceiver control via SPI. (optional)
CAN Transceiver self-diagnostics. (optional)

Table 3 Supported SWS features

Additional supported MICROSAR features
MICROSAR Identity Manager using Post-build selectable.

Table 4 Additional supported MICROSAR features

Not supported features
-

Table 5 Not supported SWS features

The CAN transceiver driver provides service functions for initialization, operation mode change and operation mode detection of the used CAN transceiver hardware. Optional service functions and callback functions are provided to detect wakeup by bus events and report them to the upper layer components.

3.2 Initialization

After power on the CAN transceiver hardware has to be initialized. Therefore the CAN transceiver driver provides two service functions.
The function `CanTrcv_30___Your_Trcv___InitMemory` initializes all necessary values for the transceiver driver. This function has to be called first after power on or reset.
The function `CanTrcv_30___Your_Trcv___Init` initializes all CAN transceiver channels which are selected by the configuration tool. Each CAN transceiver channel is switched into the operating mode: NORMAL.

CanTrcv_30___Your_Trcv___InitMemory has to be called before this function is called.
If a wakeup event was pending before the initialization the CAN transceiver driver stores this event and notifies it with the CanTrcv_30___Your_Trcv___CheckWakeup function.

3.3 Set operation mode

The operation mode of the CAN transceiver hardware is changed by service function CanTrcv_30___Your_Trcv___SetOpMode. It can be switched from normal into standby mode, from standby into sleep mode and from standby or sleep mode into normal mode. Mode change from normal into sleep mode or from sleep into standby mode is not supported by this function corresponding to the specification.

3.4 Get operation mode

To retrieve the current operation mode of a specified CAN transceiver hardware, the service function CanTrcv_30___Your_Trcv___GetOpMode has to be called.

3.5 Get version info

The service function CanTrcv_30___Your_Trcv___GetVersionInfo can be called to get the version info of the software module. This function must be enabled in the configuration tool by setting the checkbox **Version Info Api**.
The version of the CAN transceiver driver module can be acquired in two different ways. Calling the function CanTrcv_30___Your_Trcv___GetVersionInfo will return the version of the module in the structure Std_VersionInfoType which additionally includes the VendorID and the ModuleID. Accessing the version defines which are specified in the header file CanTrcv_30___Your_Trcv__.h:

AUTOSAR Revision:

CANTRCV_30___YOUR_TRCV___AR_RELEASE_MAJOR_VERSION

CANTRCV_30___YOUR_TRCV___AR_RELEASE_MINOR_VERSION

CANTRCV_30___YOUR_TRCV___AR_RELEASE_PATCH_VERSION

Module Version:

CANTRCV_30___YOUR_TRCV___SW_MAJOR_VERSION

CANTRCV_30___YOUR_TRCV___SW_MINOR_VERSION

CANTRCV_30___YOUR_TRCV___SW_PATCH_VERSION

3.6 Wakeup by bus event detection

If wakeup by bus detection is enabled in the configuration tool the callback function `CanTrcv_30___Your_Trcv___CheckWakeup` has to be called by the lower layer in case of a wakeup. This function checks the specified CAN transceiver channel. If the transceiver hardware is in sleep mode and a wakeup by bus event is detected then the function returns `E_OK`, otherwise `E_NOT_OK`.

3.6.1 Get bus wakeup reason

The service function `CanTrcv_30___Your_Trcv___GetBusWakeupReason` returns the reason which caused the wakeup.

3.6.2 Set wakeup mode

The service function `CanTrcv_30___Your_Trcv___SetWakeupMode` sets the wakeup mode which is required by the CAN interface.

3.6.3 Development Error Reporting

Development errors are reported to DET using the service `Det_ReportError`, if the pre-compile parameter `CANTRCV_30___YOUR_TRCV___DEV_ERROR_DETECT == STD_ON`.

The reported service IDs identify the services which are described in 6.1. The following table presents the service IDs and the related services:

Service ID	Service
0x00	`CanTrcv_30___Your_Trcv___Init`
0x01	`CanTrcv_30___Your_Trcv___SetOpMode`
0x02	`CanTrcv_30___Your_Trcv___GetOpMode`
0x03	`CanTrcv_30___Your_Trcv___GetBusWuReason`
0x05	`CanTrcv_30___Your_Trcv___SetWakeupMode`
0x07	`CanTrcv_30___Your_Trcv___CheckWakeup`
0x04	`CanTrcv_30___Your_Trcv___GetVersionInfo`
0x06	`CanTrcv_30___Your_Trcv___MainFunction`
0x08	`CanTrcv_30___Your_Trcv___MainFunctionDiagnostics`
0x09	`CanTrcv_30___Your_Trcv___GetTrcvSystemData`
0x0A	`CanTrcv_30___Your_Trcv___ClearTrcvWufFlag`
0x0B	`CanTrcv_30___Your_Trcv___ReadTrcvTimeoutFlag`
0x0C	`CanTrcv_30___Your_Trcv___ClearTrcvTimeoutFlag`
0x0D	`CanTrcv_30___Your_Trcv___ReadTrcvSilenceFlag`
0x0E	`CanTrcv_30___Your_Trcv___CheckWakeFlag`
0x0F	`CanTrcv_30___Your_Trcv___SetPNActivationState`

Table 6 Mapping of service IDs to services

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

Error Code		Description
0x01	CANTRCV_30___YOUR_TRCV___E_INVALID_CAN_NETWORK CANTRCV_30___YOUR_TRCV___E_INVALID_TRANSCEIVER	Invalid channel index is used in function argument list.
0x02	CANTRCV_30___YOUR_TRCV___E_PARAM_POINTER	Invalid pointer NULL_PTR is used in function argument list.
0x11	CANTRCV_30___YOUR_TRCV___E_UNINIT	CAN transceiver hardware is not initialized.
0x21	CANTRCV_30___YOUR_TRCV___E_TRCV_NOT_STANDBY	CAN transceiver hardware is not in standby mode.
0x22	CANTRCV_30___YOUR_TRCV___E_TRCV_NOT_NORMAL	CAN transceiver hardware is not in normal operation mode.
0x23	CANTRCV_30___YOUR_TRCV___E_PARAM_TRCV_WAKEUP_MODE	The requested wakeup mode is not valid.
0x24	CANTRCV_30___YOUR_TRCV___E_PARAM_TRCV_OPMODE	The requested operation mode is not supported by the underlying transceiver hardware.
0x25	CANTRCV_30___YOUR_TRCV___E_BAUDRATE_NOT_SUPPORTED	The selected baudrate is not supported by the underlying transceiver hardware.
0x40	CANTRCV_30___YOUR_TRCV___E_NO_TRCV_CONTROL	If the CAN transceiver is not under control, which means that the transceiver does remains in an invalid state, this production error is raised.

Table 7 Errors reported to DET

3.6.4 Production Code Error Reporting

Production code related errors are reported to DEM using the service `Dem_ReportErrorStatus` (specified in [3]), if the pre-compile parameter `CANTRCV_30___YOUR_TRCV___PROD_ERROR_DETECT == STD_ON`.

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

Error Code	Description
CANTRCV_30___YOUR_TRCV___E_BUS_ERROR	A physical BUS error occurred.

Table 8 Errors reported to DEM

4 Integration

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

4.1 Scope of Delivery

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

4.1.1 Static Files

File Name	Description
`CanTrcv_30___Your_Trcv__.h`	Header file which has to be included by higher layers.
`CanTrcv_30___Your_Trcv__.c`	Implementation

Table 9 Static files

4.1.2 Dynamic Files

The dynamic files are generated by the configuration tool.

File Name	Description
`CanTrcv_30___Your_Trcv___Cfg.h`	Header file contains type definitions and external data declarations. It is included by `CanTrcv_30___Your_Trcv__.h`.
`CanTrcv_30___Your_Trcv___Cfg.c`	Contains the configuration data.
`CanTrcv_GeneralTypes.h`	Header file that contains all CanTrcv types specified by SWS. This file can be included by `Can_GeneralTypes.h`.

Table 10 Generated files

4.2 Compiler Abstraction and Memory Mapping

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 that are defined for the CANTRCV and illustrates their assignment among each other.

Memory Mapping Sections \ Compiler Abstraction Definitions	CANTRCV_30___YOUR_TRCV___VAR	CANTRCV_30___YOUR_TRCV___APPL_VAR	CANTRCV_30___YOUR_TRCV___CONST	CANTRCV_30___YOUR_TRCV___CODE	CANTRCV_30___YOUR_TRCV___APPL_CODE
CANTRCV_30___YOUR_TRCV___START_SEC_CODE CANTRCV_30___YOUR_TRCV___STOP_SEC_CODE				■	■
CANTRCV_30___YOUR_TRCV___START_SEC_CONST_UNSPECIFIED CANTRCV_30___YOUR_TRCV___STOP_SEC_CONST_UNSPECIFIED			■		
CANTRCV_30___YOUR_TRCV___START_SEC_VAR_NOINIT_UNSPECIFIED CANTRCV_30___YOUR_TRCV___STOP_SEC_VAR_NOINIT_UNSPECIFIED	■	■			

Table 11 Compiler abstraction and memory mapping

4.3 Data consistency

The CAN transceiver driver calls service functions of upper layers in order to prevent interruption when accessing the CAN transceiver pins.

These service functions have to be provided by the components VStdLib, Schedule Manager or OSEK OS depending on which of these components is used for interrupt disable/restore handling. The component for interrupt control handling has to be selected in the configuration tool.

4.4 Integration

The driver has to be extended by hardware specific functionality. Within `CanTrcv_30___Your_Trcv__.c` and `CanTrcv_30___Your_Trcv__.h`, the parts that have to be implemented are marked with the following token:

<Your_Trcv_Code>

Please refer to the comments at the dedicated code part for integration tips. Also refer to the following chapters for details.

Info

Replace the placeholders `__Your_Trcv__` and `__YOUR_TRCV__` with the according name of the used transceiver. This procedure has to be done in both, the source code and the BSWMD files.

`__YOUR_TRCV__` is used for definitions in upper case (e.g. TJA1041).

`__Your_Trcv__` is used for variables in camel case (e.g. Tja1041).

4.5 Timers

As the underlying transceiver hardware may have some time constraints that must be met, the transceiver driver sometimes needs to wait some time until the next request to the hardware can be made.

An application function which handles this wait states has to be implemented by the user. To enable or disable this callback function and all predefined timers, you have to specify the following constant in the `CanTrcv_30___Your_Trcv__.h`:

```
# define CANTRCV_30___YOUR_TRCV___USE_TIMERS                    STD_ON
```

The prototype of the function must be as follows:

```
FUNC(void, CANTRCV_30___YOUR_TRCV___CODE) Appl_CanTrcv_30___Your_Trcv___Wait(uint8
TimerIndex);
```

The parameter `TimerIndex` is used to distinguish between the timers needed by the transceiver driver. `TimerIndex` is represented by a symbolic constant that is defined in the CanTrcv_30___Your_Trcv__.h. The following table lists all available timer indexes:

Timer Index (symbolic constant)	Wait Time	Description
`kCanTrcv_30___Your_Trcv___LoopInit`	> 50µs	This timer is called by the transceiver driver in function `Init()` after the transition to normal mode was made and before switching to the mode, specified in the configuration tool.

Table 12 Timer indexes and their wait times

Example of the implementation

```
FUNC(void, CANTRCV_30___YOUR_TRCV___CODE)
Appl_CanTrcv_30___Your_Trcv___Wait(uint8 TimerIndex)
{
    uint32 timer;

    switch (TimerIndex)
    {
      case kCanTrcv_30___Your_Trcv___LoopInit:
        timer = 100;
        break;

      default:
        timer = 0;
        break;
    }

    for (timer; timer != 0; --timer)
       /* nothing to do */;
}
```

4.6 Generated Data

The configuration tool generates various tables that can be used within the implementation. Also refer to the following files:

`CanTrcv_30___Your_Trcv___Cfg.h` (Type definitions, declarations)

`CanTrcv_30___Your_Trcv___Cfg.c` (Initializers, definitions)

4.6.1 Access to generic data

The access to generated data is abstracted by C-macros. Please use only them to access generic data for each channel. There are following macros provided.

`CanTrcvCfg_GetInitState(Channel)`	State that shall be reached after CanTrcv_Init() has finished.
`CanTrcvCfg_IsWakeupByBusUsed(Channel)` **Return type:** `boolean`	TRUE if Wakeup By Bus over CanTrcv is activated for the given channel, FALSE otherwise.
`CanTrcvCfg_IsChannelUsed(Channel)` **Return type:** `boolean`	TRUE if the channel has to be used, FALSE otherwise.
`CanTrcvCfg_GetWakeupSource(Channel)` **Return type:** `EcuM_WakeupSourceType`	Wakeup Source of the given channel that will be reported to EcuM after a wakeup was detected.
Only if `CANTRCV_30___YOUR_TRCV___USE_ICU == TRUE`:	
`CanTrcvCfg_IsIcuChannelSet(Channel)` **Return type:** `boolean`	TRUE if an ICU channel has been configured for the given channel, FALSE otherwise.
`CanTrcvCfg_GetIcuChannel(Channel)` **Return type:** `Icu_ChannelType`	Configured ICU channel for the given CanTrcv channel. Only valid if IcuChannelSet == TRUE.
Only for DIO Interface:	
`CanTrcv_30___Your_Trcv___Get<NameOfPin>OfDioConfiguration(Channel)` **Return type:** `Dio_ChannelType`	For each configured DIO pin, one member is generated. The value of this Member is <Pin>_<Channel Index> in upper case. (E.g. DIO_PIN_TRCV_RX_0).

Table 13 Access to generic data

The unmodified template driver uses all members with exception of “Dio_ChannelType <Pin>”. Except for the DIO Pins, no code has to be implemented.

Note
When using SPI interface all sequences / channels for a given channel must be generated “by hand” e.g. by using a static table within the driver code. There is no tool support to configure any reference to a SPI sequence / channel / job.

4.6.2 Partial networking data

In case of the underlying CanTrcv supports selective wakeup then there are following additional C-macros provided to access the related generated partial networking data. Please use only these C-macros to access the generated data for each channel

`CanTrcv_30___Your_Trcv___GetCanIdOfPnCfg(Channel)` **Return type:** `uint32`	Contains the CAN ID as entered in the corresponding field in the generation tool.
`CanTrcv_30___Your_Trcv___GetCanIdMaskOfPnCfg(Channel)` **Return type:** `uint32`	Contains the CAN ID MASK as entered in the corresponding field in the generation tool.

`CanTrcv_30___Your_Trcv___GetDataMask0OfPnPayloadCfg(Channel)`
`CanTrcv_30___Your_Trcv___GetDataMask1OfPnPayloadCfg(Channel)`
`CanTrcv_30___Your_Trcv___GetDataMask2OfPnPayloadCfg(Channel)`
`CanTrcv_30___Your_Trcv___GetDataMask3OfPnPayloadCfg(Channel)`
`CanTrcv_30___Your_Trcv___GetDataMask4OfPnPayloadCfg(Channel)`
`CanTrcv_30___Your_Trcv___GetDataMask5OfPnPayloadCfg(Channel)`
`CanTrcv_30___Your_Trcv___GetDataMask6OfPnPayloadCfg(Channel)`
`CanTrcv_30___Your_Trcv___GetDataMask7OfPnPayloadCfg(Channel)`
Return type: `uint8`
	Contains the DATA bytes. One C-macro for each data element.

`CanTrcv_30___Your_Trcv___GetDlcOfPnCfg(Channel)` **Return type:** `uint8`	Contains the configured DLC.
`CanTrcv_30___Your_Trcv___IsExtendedCanIdOfPnCfg(Channel)` **Return type:** `uint8`	Contains value TRUE if the CAN ID/ CAN ID MASK shall be interpreted as extended one otherwise FALSE.
`CanTrcv_30___Your_Trcv___GetBaudrateOfPnCfg(Channel)` **Return type:** `uint8`	Returns the configured baudrate [KBit/s] (e.g. 125, 250, 500, …)
`CanTrcvCfg_IsPnEnabled(Channel)` **Return type:** `boolean`	Indicates per channel whether selective wakeup is enabled or not. TRUE: enabled FALSE: disabled

`CanTrcvCfg_GetWakeupSourceSyserr(Channel)`
Return type:
`EcuM_WakeupSourceType`
	Returns the configured “CanTrcvSyserrWakeupSourceRef“

`CanTrcvCfg_GetWakeupSourcePor(Channel)`
Return type:
`EcuM_WakeupSourceType`
	Returns the configured “CanTrcvPorWakeupSourceRef”

Table 14 Partial networking data

None of these values are used in the unmodified template driver. It is up to the user to implement the code necessary to download the data to the underlying CanTrcv hardware. The data must be written at `Init_Channel_PnData` in `CanTrcv_30___Your_Trcv__.c`.

4.7 Handling of asynchronous APIs

Please note
This chapter only affects CanTrcv hardware that uses an asynchronous SPI Interface. Transceivers with DIO interface are always synchronous and thus do not need any special behavior.

Due to the complexity of the implementation it is NOT RECOMMENDED to use asynchronous SPI access.

Throughout this chapter, CanTrcv_* instead of CanTrcv_30___Your_Trcv__* is used in order to improve readability.

When using asynchronous SPI access, all APIs except `CanTrcv_Init` shall behave asynchronously. This usually means that a call to an API only *requests* the dedicated action but does not actually *perform* it. Instead the request shall be held pending until the requested action is either completely performed or another concurrent action is requested.

As there are no restrictions basically each request can be interrupted by another request. The driver must thus handle concurrent and interrupted requests in order to guarantee proper functionality.

It is also required by the driver to inform upper layers about finished or cancelled requests according to the SWS.

The template driver contains a few mechanisms to help the user to implement a correct driver. Depending on the underlying CanTrcv hardware the actual requirements for implementation varies.

4.7.1 Request handling

The template driver already implements a mechanism to handle requests in a correct way. If asynchronous SPI access is used, the template driver uses `CanTrcv_Set*Req` to store requests. These requests can be queried with the `CanTrcv_Get*Req` macros. Refer to `CanTrcv_30___Your_Trcv__.c` for details.

Example:

If CanTrcv_SetOpMode is called, the template driver stores the request with `CanTrcv_SetOpModeReq`. In context of `CanTrcv_MainFunction` the user implementation has to check the pending request with `CanTrcv_GetOpModeReq` and has to perform all necessary tasks to complete it.

4.7.1.1 Concurrent operation modes

It is required that always the last pending operation mode request is completed. If another operation mode request is currently pending the driver has to safely cancel this old request before completing the new request.

Higher layer has to be informed only for the last operation mode request. Thus the implementation has to take care that no invalid indication is called.

The template driver takes care about this in a limited manner. To query a new request the user has to implement `SetOpModeNormal`, `SetOpModeStandby` and `SetOpModeSleep` in `CanTrcv_30___Your_Trcv__.c`.

Within `CanTrcv_SetOpMode` it is NOT ALLOWED to perform any SPI request. Instead all hardware access must be done in context of `CanTrcv_MainFunction`.

As soon as the request is finished the internal callback `CanTrcv_TrcvModeIndication` MUST be called by the user implementation.

The following table illustrates the necessary steps to change an operation mode.

Requested Mode	Actions
NORMAL	Call `CanTrcv_SetConfirmPnAvaibilityRequest` Read the status flags (via GetStatusFlags) Call `CanTrcv_GetStatusFlagsIndication` as soon as the status flags are read. Perform the actual mode change Call `CanTrcv_TrcvModeIndication` as soon as the mode change is complete
STANDBY	Perform the actual mode change Call `CanTrcv_TrcvModeIndication` as soon as the mode change is complete
SLEEP	Perform the actual mode change Call `CanTrcv_TrcvModeIndication` as soon as the mode change is complete

Table 15 Change of Operation Mode

If a mode change request is not yet completed when another new mode is requested the template driver executes the code located at `CancelOpModeReq`. Here, the user has to implement a function to safely cancel the old request. If it is necessary to wait for any event (e.g. SPI Indication) `CanTrcv_MainFunction` must handle the cancel request.

While a request is marked as “to be cancelled” the user implementation has to take care that the internal callback `CanTrcv_TrcvModeIndication` is NOT being called.

As soon as the old request is cancelled, the user implementation MUST call the macro `CanTrcv_ProcessNextRequest`. And check if a new request is pending.

If an action other than mode change is requested the user implementation must handle the other request AFTER the mode change request has completed.

If another action is currently performed while a mode change is requested the user implementation must handle the operation mode change AFTER the other request has completed.

4.7.1.2 Request to CheckWakeFlag / ClearTrcvWufFlag / CB_WakeupByBus

If a request to `CheckWakeFlag` or `ClearTrcvWufFlag` is pending, the corresponding code in `CanTrcv_30__Your_Trcv__.c` is executed.

The user implementation MUST NOT ignore any of these requests. If there is more than one request of the same type pending, the confirmation must be done only once.

Within the APIs it is not allowed to use SPI access. Instead the handling must be done in context of `CanTrcv_Mainfunction`.

As soon as a request has completed the user implementation MUST call the corresponding internal callback (`CanTrcv_CheckWakeFlagIndication` / `CanTrcv_ClearTrcvWufFlagIndication` / `CanTrcv_CbWakeupByBusIndication`).

If the hardware implementation allows handling multiple requests with the same SPI communication, it is allowed to call multiple internal callbacks at the same time.

4.7.1.3 Status Flag handling

When status flags are queried (at GetStatusFlags in `CanTrcv__30___Your_Trcv__.c`), the driver has to read out all status flags of the underlying CanTrcv hardware as soon as possible.

Upon request, the current status flags have to be marked as invalid by setting `CanTrcv_30___Your_Trcv___Prob[index].statusFlagsRdy = FALSE.`

If the SPI is currently blocked by another request, the request to read the status flags shall be held pending.

As soon as the status flags are completely read, the user implementation has to call the internal callback `CanTrcv_GetStatusFlagsIndication`.

4.7.1.4 Indication handling

Most internal callbacks can be executed directly from a SPI callback. Depending on the last executed SPI command, the pending request and the state of the driver, the corresponding callback should be called.

Below is an example of the Elmos E520.13 implementation:

```
FUNC(void, CANTRCV_30_E52013_CODE) CanTrcv_30_E52013_SpiIndication(uint8 CanTrcvIndex)
{
  uint8 event = CANTRCV_30_E52013_WORKER_EV_NONE;
  switch(CanTrcv_30_E52013SpiCmdBuffer[CanTrcvIndex])
  {
  case CANTRCV_30_E52013_RD_SYS_DIAG:
    event = CANTRCV_30_E52013_WORKER_EV_RD_SYS_DIAG;
    break;

  case CANTRCV_30_E52013_RD_SYS_SET:
    event = CANTRCV_30_E52013_WORKER_EV_RD_SYS_SET;
    break;

  case CANTRCV_30_E52013_WR_SYS_SET:
    event = CANTRCV_30_E52013_WORKER_EV_WR_SYS_SET;
    break;

  default:
    break;
  }

  if (event != CANTRCV_30_E52013_WORKER_EV_NONE)
  {
    if (event == CANTRCV_30_E52013_WORKER_EV_RD_SYS_DIAG)
    {
      /* New status flags received... */
      CanTrcv_30_E52013_GetStatusFlagsIndication(CanTrcvIndex);
    }
    /* ModeWorker */
    if (   (CanTrcv_30_E52013_Prob[CanTrcvIndex].WorkerState != CANTRCV_30_E52013_WORKER_STOPPED)
        && (CanTrcv_30_E52013_Prob[CanTrcvIndex].isInit == CANTRCV_30_E52013_IS_INIT)
        )
    {
      /* Inform worker about this event */
      (void)CanTrcv_30_E52013_ModeWorker(CanTrcvIndex, event);
    }

    /* -- Indications */
    if (event == CANTRCV_30_E52013_WORKER_EV_RD_SYS_DIAG)
    {
      if (CanTrcv_30_E52013_GetClrWufFlagReq(CanTrcvIndex) != 0x0u)
      {
        CanTrcv_30_E52013_ClearTrcvWufFlagIndication(CanTrcvIndex);
      }

      if (CanTrcv_30_E52013_IsReqFlagPnd(CanTrcvIndex))
      {
        if (CanTrcv_30_E52013_GetChkWakFlagReq(CanTrcvIndex) != 0x0u)
        {
          CanTrcv_30_E52013_CheckWakeFlagIndication(CanTrcvIndex);
        }
        if (CanTrcv_30_E52013_GetCbByBusFlagReq(CanTrcvIndex) != 0x0u)
        {
          CanTrcv_30_E52013_CbWakeupByBusIndication(CanTrcvIndex);
        }
        /* Release RdDiag request here as new requests to read DIAG should be queried */
        CanTrcv_30_E52013_Prob[CanTrcvIndex].ReadDiagRequestState =
CANTRCV_30_E52013_RD_DIAG_STATE_IDLE;
      }
    }
  }
  /* Free SPI */
  CanTrcv_30_E52013SpiCmdBuffer[CanTrcvIndex] = CANTRCV_30_E52013_NO_REQ;
}
```

5 Dependencies to other components

5.1 Dio driver

Depending on the configuration, the CanTrcv driver performs hardware access by calling service functions of the lower layer component Dio driver:

> Function Dio_WriteChannel() is used to set the logical level of the channel pins to which the CAN transceiver hardware is connected.

> Function Dio_ReadChannel() is used to get the logical level of the channel pins to which the CAN transceiver hardware is connected.

> The Dio driver has to provide the channel dependent assignment for the CAN transceiver hardware pins which are specified in the tool. These pins are referred by the CAN transceiver driver by using the symbolic names specified in the tool.

5.2 SPI driver

Depending on the configuration, the CanTrcv driver performs hardware access by calling service functions of the lower layer component SPI driver:

- Function `Spi_SyncTransmit()` / `Spi_AsyncTransmit()` is used to issue a sequence in order to read from or write to the transceiver hardware.
- The SPI driver must be configured to allow access to the transceiver. This means that sequences, jobs and channels must be defined prior using the transceiver driver.

5.2.1 SPI Configuration

The configuration of the SPI driver strongly depends on the underlying CanTrcv hardware. There is no tool support for specifying which sequences / channels / jobs are used by the CanTrcv driver.

5.3 Icu driver

The CAN transceiver driver performs hardware access by calling service functions of the lower layer component Icu driver:

- Function `Icu_DisableNotification` is used by the transceiver driver to disable the ICU notification after wakeup event notification.
- Function `Icu_EnableNotification` is used by the transceiver driver to enable the ICU notification during the transition into the STANDBY mode.

Please note
The following chapter applies only to this use case:
- the used CAN controller does not support the feature "wakeup by CAN bus", i.e. the CAN controller cannot detect incoming CAN messages and generate a so-called wakeup-interrupt
- the ECU shall wake up and start its own communication due to detected communication on the CAN bus

The proposal described in this chapter enables the user to support the use-case by using an additional µC I/O port to generate the wakeup information via the ICU driver. This I/ O port is connected either parallel to the CAN Rx port or is directly attached to the CAN transceivers ERR port (depending on the used CAN transceiver). The following steps have to be done for every CAN channel that shall be able to wake up via the CAN bus:

5.3.1 ICU configuration

Add an ICU channel

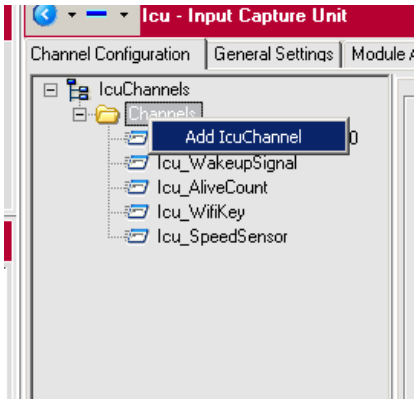

Figure 5-1 Add an ICU channel

The user has to configure the ICU channel and to add a signal notification function.

Additionally the wakeup source for the CAN channel which shall be handled via this ICU channel have to be chosen.

In dependency of the provided ICU configuration options it is possible to configure the “IcuDefaultStartEdge”. This option should be configured to ICU_FALLING_EDGE, because

the wakeup event is provided by the Rx pin and/ or the ERR pin via a level change from recessive to dominant.

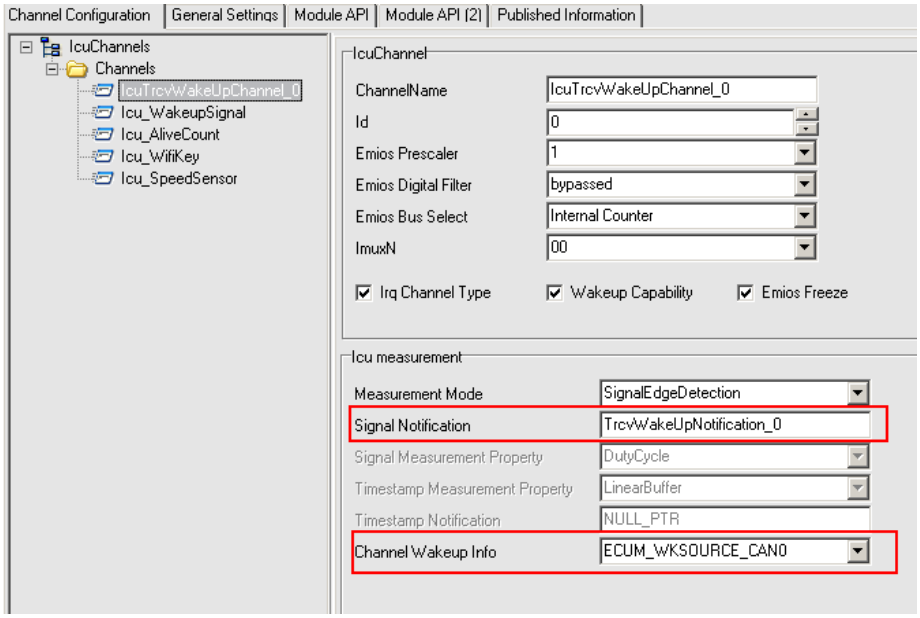

Figure 5-2 ICU channel configuration for wakeup via transceiver

If the transceiver shall also wake up the ECU and not only the CAN channel then it is necessary to activate the “Wakeup Capability” for this ICU channel.

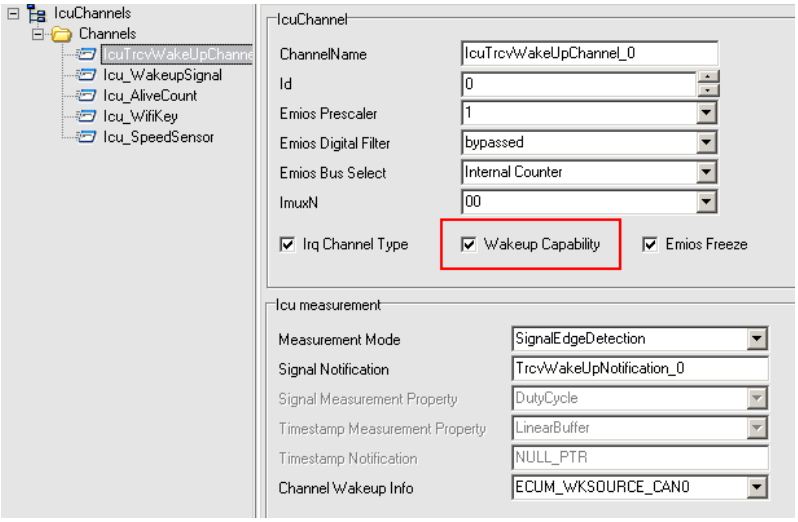

Figure 5-3 ICU wakeup capability

5.3.2 Implementation of the signal notification function

The user has to implement the signal notification function as shown in the following example:

Example

```
void Icu_TrcvWakeUpNotification_0(void)
{
  /* inform the EcuM about the wakeup event, the parameter
     is the configured transceiver wakeup source */
  EcuM_CheckWakeup(ECUM_WKSOURCE_CAN0);
}
```

Please note

If no wakeup validation is used for the configured wakeup source, then it is possible to call `EcuM_SetWakeupEvent()` instead of `EcuM_CheckWakeup()`.

6 API Description

6.1 Services provided by CANTRCV

The CANTRCV API consists of services, which are realized by function calls.

6.1.1 CanTrcv_30___Your_Trcv___InitMemory

Prototype	
`void CanTrcv_30___Your_Trcv___InitMemory ( void )`	
Parameter	
-	-
Return code	
-	-
Functional Description	
This function initializes the memory and needed values of the CAN transceiver driver.	
Particularities and Limitations	
> This function must be called before any other functionality of the CAN transceiver driver.	
Expected Caller Context	
> This function must be called from task level and is not reentrant.	

Table 16 CanTrcv_30___Your_Trcv___InitMemory

6.1.2 CanTrcv_30___Your_Trcv___Init

Prototype	
`void CanTrcv_30___Your_Trcv___Init ( CanTrcv_30___Your_Trcv___ConfigType* ConfigPtr )`	
Parameter	
`ConfigPtr`	Pointer to the `CanTrcv_30___Your_Trcv___Config` struct. If multiple configurations are available, the active configuration can be selected by using the related `CanTrcv_30___Your_Trcv___Config_<IdentityName>_Ptr` struct
Return code	
-	-
Functional Description	
> This function initializes all channels of the CAN Transceiver driver which are configured in the configuration tool.	
Particularities and Limitations	
> The function `CanTrcv_30___Your_Trcv___InitMemory` must be called before the function `CanTrcv_30___Your_Trcv___Init` can be called. > This function must be called before any other service functionality of the Transceiver driver.	
Expected Caller Context	
> This function must be called from task level and is not reentrant.	

Table 17 CanTrcv_30___Your_Trcv___Init

6.1.3 CanTrcv_30___Your_Trcv___SetOpMode

Prototype	
`Std_ReturnType` **`CanTrcv_30___Your_Trcv___SetOpMode`** `(uint8 CanTrcvIndex, CanTrcv_TrcvModeType OpMode )`	
Parameter	
OpMode	Pointer which contains the desired operation mode.
CanTrcvIndex	Index of the selected transceiver.
Return code	
E_OK / E_NOT_OK	E_OK: is returned if the transceiver state has been changed to the requested mode. E_NOT_OK: is returned if the transceiver state change has failed or the parameter is out of the allowed range. The previous state has not been changed.
Functional Description	
Sets the CAN transceiver to the requested operation mode. These operation modes are: ■ `CANTRCV_TRCVMODE_NORMAL` ■ `CANTRCV_TRCVMODE_STANDBY` ■ `CANTRCV_TRCVMODE_SLEEP` If the transceiver is requested to change into NORMAL mode and selective wakeup is enabled on channel `CanTrcvIndex` the hardware is queried for error flags. If no error was detected the driver calls the notification function `CanIf_30___Your_Trcv___ConfirmPnAvailability` in order to confirm that selective wakeup feature is available. If return code is E_OK the notification function `CanIf_30___Your_Trcv___TrcvModIndication` will be called if mode change is completed. Note that the notification may occur in interrupt context, in task context or in context of `CanTrcv_30___Your_Trcv___SetOpMode`. Also, it is not ensured that the mode notified to `CanIf_30___Your_Trcv___TrcvModeIndication` matches the requested mode.	
Particularities and Limitations	
> The CAN transceiver driver must be initialized. > Not each transition from one mode to any other is allowed. For these limitations please refer to chapter Set operation mode.	
Expected Caller Context	
> This function can be called from task or interrupt level and is not reentrant.	

Table 18 CanTrcv_30___Your_Trcv___SetOpMode

6.1.4 CanTrcv_30___Your_Trcv___GetOpMode

Prototype	
`Std_ReturnType` **`CanTrcv_30___Your_Trcv___GetOpMode`** `(uint8 CanTrcvIndex, CanTrcv_TrcvModeType *OpMode )`	
Parameter	
OpMode	Pointer to operation mode of the bus the API is applied to.
CanTrcvIndex	Index of the selected transceiver.
Return code	
E_OK / E_NOT_OK	E_OK: is returned if the operation mode was detected. E_NOT_OK: is returned if the operation mode was not detected.
Functional Description	
Stores the current operation mode of the selected CAN transceiver to `OpMode`. These operation modes are: > `CANTRCV_TRCVMODE_NORMAL` > `CANTRCV_TRCVMODE_STANDBY` > `CANTRCV_TRCVMODE_SLEEP`	
Particularities and Limitations	
> The CAN transceiver driver must be initialized. > If a mode change was requested before, the reported operation mode may not be valid until the mode change is completed and `CanIf_30___Your_Trcv___TrcvModIndication` was called.	
Expected Caller Context	
> This function can be called from task or interrupt level and is not reentrant.	

Table 19 CanTrcv_30___Your_Trcv___GetOpMode

6.1.5 CanTrcv_30___Your_Trcv___GetBusWuReason

Prototype	
`Std_ReturnType` **`CanTrcv_30___Your_Trcv___GetBusWuReason`** `(uint8 CanTrcvIndex, CanTrcv_TrcvWakeupReasonType *Reason )`	
Parameter	
CanTrcvIndex	Index of the selected transceiver.
Reason	Pointer to wake up reason of the bus the API is applied to.
Return code	
E_OK / E_NOT_OK	E_OK: is returned if the wake up reason was detected. E_NOT_OK: is returned if the wake up reason was not detected.
Functional Description	
Stores the last the wakeup reason for the channel `CanTrcvIndex` to `Reason`. These wakeup reasons are: > `CANTRCV_WU_INTERNALLY`: The wakeup was caused by setting the CAN transceiver to normal operation mode via `CanTrcv_30___Your_Trcv___SetOpMode`. > `CANTRCV_WU_ERROR`: No wakeup was detected by the transceiver and no reason is stored. The function returns `E_NOT_OK`. > `CANTRCV_WU_NOT_SUPPORTED`: No wakeup detection supported by this CAN transceiver. The function returns `E_NOT_OK`. > `CANTRCV_WU_BY_BUS`: The wakeup was caused by an external bus wakeup. > `CANTRCV_WU_RESET`: The wakeup was detected after a reset. > `CANTRCV_WU_POWER_ON`: The transceiver detected a power-on wakeup > `CANTRCV_WU_BY_SYSERR`: An internal hardware error occurred that caused the transceiver to wake up. > `CANTRCV_WU_BY_PIN`: The wakeup was caused by applying an edge to the WAKE pin.	
Particularities and Limitations	
> The CAN transceiver driver must be initialized. > The wakeup reason represents always the last detected wakeup reason. If there was more than one wakeup detected only the last one will be reported.	
Expected Caller Context	
> This function can be called from task or interrupt level and is not reentrant.	

Table 20 CanTrcv_30___Your_Trcv___GetBusWuReason

6.1.6 CanTrcv_30___Your_Trcv___SetWakeupMode

Prototype	
`Std_ReturnType CanTrcv_30___Your_Trcv___SetWakeupMode (uint8 CanTrcvIndex , CanTrcv_TrcvWakeupModeType TrcvWakeupMode )`	
Parameter	
TrcvWakeupMode	Requested transceiver wakeup reason.
CanTrcvIndex	Index of the selected transceiver.
Return code	
E_OK / E_NOT_OK	E_OK: is returned, if the wakeup state has been changed to the requested mode. E_NOT_OK: is returned, if the wakeup state change has failed or the parameter is out of the allowed range. The previous state has not been changed.
Functional Description	
Enables, disables or clears reporting of wakeup events on channel `CanTrcvIndex`. > `CANTRCV_WUMODE_ENABLE`: Report wakeup events to upper layer. > `CANTRCV_WUMODE_DISABLE`: Do not report wakeup events to upper layer. > `CANTRCV_WUMODE_CLEAR`: Clear a pending wakeup event.	
Particularities and Limitations	
> The CAN transceiver driver must be initialized. > If the wakeup handling is not enabled in GENy the function always return E_NOT_OK.	
Expected Caller Context	
> This function is called from task or interrupt level and not reentrant.	

Table 21 CanTrcv_30___Your_Trcv___SetWakeupMode

6.1.7 CanTrcv_30___Your_Trcv___GetVersionInfo

Prototype	
`void CanTrcv_30___Your_Trcv___GetVersionInfo (Std_VersionInfoType VersionInfo)`	
Parameter	
VersionInfo	Pointer to version information of this module.
Return code	
-	-
Functional Description	
Gets the version of the module and returns it in VersionInfo.	
Particularities and Limitations	
> The CAN transceiver driver must be initialized.	
Expected Caller Context	
> This function can be called from task or interrupt level and is not reentrant.	

Table 22 CanTrcv_30___Your_Trcv___GetVersionInfo

6.1.8 CanTrcv_30___Your_Trcv___CheckWakeup

Prototype	
`Std_ReturnType: CanTrcv_30___Your_Trcv___CheckWakeup (uint8 CanTrcvIndex)`	
Parameter	
CanTrcvIndex	Index of the selected transceiver.
Return code	
`E_OK / E_NOT_OK`	In synchronous mode: `E_OK` a wakeup-by-bus event was detected `E_NOT_OK` no wakeup was detected or an error occurred In asynchronous mode: `E_OK`: the request to check for wakeup was acknowledged. `E_NOT_OK`: an error occurred.
Functional Description	
This function requests the CanTrcv driver to check for wakeups and report them. If a wakeup was detected, the CanTrcv reports it by calling of `EcuM_SetWakeupEvent`. In asynchronous mode, if no wakeup was detected `EcuM_EndCheckWakeup` is called.	
Particularities and Limitations	
> The CAN transceiver driver must be initialized. > Do not use return value `E_OK` for wakeup detection. A wakeup can be considered as valid only if `EcuM_SetWakeupEvent` was called for the corresponding wakeup source.	
Expected Caller Context	
> This function can be called from task or interrupt level and is not reentrant.	

Table 23 CanTrcv_30___Your_Trcv___CB_WakeupByBus

6.1.9 CanTrcv_30___Your_Trcv___GetTrcvSystemData

Prototype	
`Std_ReturnType` **`CanTrcv_30___Your_Trcv___GetTrcvSystemData`** `(uint8 CanTrcvIndex, uint32* TrcvSysData)`	
Parameter	
`CanTrcvIndex`	Index of the selected transceiver
`TrcvSysData`	Pointer to the diagnosis data buffer to store the information in.
Return code	
`E_OK` / `E_NOT_OK`	`E_OK` if the diagnosis register was successfully read, `E_NOT_OK` otherwise.
Functional Description	
Reads the diagnosis registers of the underlying transceiver hardware and stores them in `TrcvSysData`.	
Particularities and Limitations	
■ The CAN transceiver driver must be initialized. ■ The underlying transceiver hardware must support Selective Wakeup / Partial Networking ■ `SysDiagData` contains only valid information if `E_OK` was returned. ■ The function will not accept any requests if there are still SPI requests pending. In this case `E_NOT_OK` is returned.	
Expected Caller Context	
This function can be called from task or interrupt level and is not reentrant.	

Table 24 CanTrcv_30___Your_Trcv___GetTrcvSystemData

6.1.10 CanTrcv_30___Your_Trcv___ClearTrcvWufFlag

Prototype	
`Std_ReturnType` **`CanTrcv_30___Your_Trcv___ClearTrcvWufFlag`** `(uint8 CanTrcvIndex)`	
Parameter	
`CanTrcvIndex`	Index of the selected transceiver
Return code	
`E_OK` / `E_NOT_OK`	`E_OK` if the WUF flag has been cleared, `E_NOT_OK` otherwise.
Functional Description	
This service requests the transceiver driver to clear all wakeup flags from the underlying transceiver hardware. If E_OK is returned, the driver will call the notification function `CanIf_30___Your_Trcv___ClearTrcvWufFlagIndication` as soon as the request has been completed. Note that the notification may occur in interrupt context, in task context or in context of `CanTrcv_30___Your_Trcv___ClearTrcvWufFlag`.	
Particularities and Limitations	
■ The CAN transceiver driver must be initialized. ■ The underlying transceiver hardware must support Selective Wakeup / Partial Networking	
Expected Caller Context	
This function can be called from task or interrupt level and is not reentrant.	

Table 25 CanTrcv_30___Your_Trcv___ClearTrcvWufFlag

6.1.11 CanTrcv_30___Your_Trcv___ReadTrcvTimeoutFlag

Prototype	
`Std_ReturnType` **`CanTrcv_30___Your_Trcv___ReadTrcvTimeoutFlag`** `(uint8 CanTrcvIndex, CanTrcv_30___Your_Trcv___FlagStateType* FlagState)`	
Parameter	
`CanTrcvIndex`	Index of the selected transceiver
`FlagState`	State of the timeout flag.
Return code	
`E_OK` / `E_NOT_OK`	`E_OK` if status of the timeout flag is successfully read. `E_NOT_OK` otherwise.
Functional Description	
Reads the status of the timeout flag from the underlying transceiver hardware and stores it to `FlagState`.	
Particularities and Limitations	
■ The CAN transceiver driver must be initialized.	
Expected Caller Context	
This function can be called from task or interrupt level and is not reentrant.	

Table 26 CanTrcv_30___Your_Trcv___ReadTrcvTimeoutFlag

6.1.12 CanTrcv_30___Your_Trcv___ClearTrcvTimeoutFlag

Prototype	
`Std_ReturnType` **`CanTrcv_30___Your_Trcv___ClearTrcvTimeoutFlag`** `(uint8 CanTrcvIndex)`	
Parameter	
`CanTrcvIndex`	Index of the selected transceiver
Return code	
`E_OK` / `E_NOT_OK`	`E_OK` if the timeout flag has been cleared, `E_NOT_OK` otherwise.
Functional Description	
Clears the timeout flag from the underlying transceiver hardware.	
Particularities and Limitations	
■ The CAN transceiver driver must be initialized.	
Expected Caller Context	
This function can be called from task or interrupt level and is not reentrant.	

Table 27 CanTrcv_30___Your_Trcv___ClearTrcvTimeoutFlag

6.1.13 CanTrcv_30___Your_Trcv___ReadTrcvSilenceFlag

Prototype	
`Std_ReturnType` **`CanTrcv_30___Your_Trcv___ReadTrcvSilenceFlag`** `(uint8 CanTrcvIndex, CanTrcv_30___Your_Trcv___FlagStateType* FlagState)`	
Parameter	
`CanTrcvIndex`	Index of the selected transceiver
`FlagState`	State of the silence flag.
Return code	
`E_OK` / `E_NOT_OK`	`E_OK` if status of the silence flag is successfully read. `E_NOT_OK` otherwise.
Functional Description	
Reads the status of the silence flag from the underlying transceiver hardware and stores it to `FlagState`.	
Particularities and Limitations	
■ The CAN transceiver driver must be initialized.	
Expected Caller Context	
This function can be called from task or interrupt level and is not reentrant.	

Table 28 CanTrcv_30___Your_Trcv___ReadTrcvSilenceFlag

6.1.14 CanTrcv_30___Your_Trcv___CheckWakeFlag

Prototype	
`Std_ReturnType` **`CanTrcv_30___Your_Trcv___CheckWakeFlag`** `(uint8 CanTrcvIndex)`	
Parameter	
`CanTrcvIndex`	Index of the selected transceiver
Return code	
`E_OK` / `E_NOT_OK`	`E_OK` if request for checking wakeup flag has been accepted, `E_NOT_OK` otherwise.
Functional Description	
Requests the driver to check the status of the wake flag of the underlying transceiver hardware. Any detected wakeup will be reported through service EcuM_SetWakeupEvent. The correct wakeup reason can be determined by using `CanTrcv_30___Your_Trcv___GetBusWuReason`. If E_OK is returned, the driver will call the notification function `CanIf_30___Your_Trcv___CheckTrcvWakeFlagIndication` as soon as the request has been completed. Note that the notification may occur in interrupt context, in task context or in context of `CanTrcv_30___Your_Trcv___CheckWakeFlag`.	
Particularities and Limitations	
■ The CAN transceiver driver must be initialized.	
Expected Caller Context	
This function can be called from task or interrupt level and is not reentrant.	

Table 29 CanTrcv_30___Your_Trcv___CheckWakeFlag

6.1.15 CanTrcv_30___Your_Trcv___MainFunction

Prototype	
`void CanTrcv_30___Your_Trcv___MainFunction (void)`	
Parameter	
-	-
Return code	
-	-
Functional Description	
This service can be called from task level and periodically checks if a wakeup was detected by the underlying transceiver hardware.	
Particularities and Limitations	
■ The CAN transceiver driver must be initialized.	
Expected Caller Context	
This function can be called from task context and is not reentrant.	

Table 30 CanTrcv_30___Your_Trcv___MainFunction

6.1.16 CanTrcv_30___Your_Trcv___SetPNActivationState

Prototype	
`Std_ReturnType CanTrcv_30__Your_Trcv__SetPNActivationState (CanTrcv_30___Your_Trcv___PNActivationType ActivationState)`	
Parameter	
`ActivationState`	Specifies whether to enable or disable selective wakeup.
Return code	
`E_OK / E_NOT_OK`	`E_OK` is returned if PN activation state was successfully changed. `E_NOT_OK` is returned if PN activation state could not be changed or an error occurred.
Functional Description	
This service changes the state of the selective wakeup feature. If called with `PN_ENABLED` CanTrcv hardware will be configured to wake up on configured frames only (WUF). If called with `PN_DISABLED` CanTrcv hardware will be configured to wake up on any bus activity (WUP). In order to enable selective wakeup it must be also configured as enabled in generation tool.	
Particularities and Limitations	
■ The CAN transceiver driver must be initialized. ■ State change to `PN_DISABLED` will be effective after next mode change. ■ Selective wakeup can be enabled only for channels that have PN activated in generation tool. ■ It is strongly recommended to use this API only directly after `CanTrcv_30___Your_Trcv___Init()`.	
Expected Caller Context	
This function must be called from task and is not reentrant. The API must not be interrupted by any other CanTrcv API.	

Table 31 CanTrcv_30___Your_Trcv___SetPNActivationState

6.2 Services used by CANTRCV

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

Component	API
DET	Det_ReportError
DEM	Dem_SetEventStatus
SPI	Spi_SetupEB
SPI	Spi_SyncTransmit
SPI	Spi_AsyncTransmit
DIO	Dio_WriteChannel
DIO	Dio_ReadChannel
CANIF	CanIf_TrcvModeIndication
CANIF	CanIf_ConfirmPnAvailability
CANIF	CanIf_ClearTrcvWufFlagIndication
CANIF	CanIf_CheckTrcvWakeFlagIndication
CANIF	CanIf_TrcvModeIndication
ICU	Icu_EnableNotification
ICU	Icu_DisableNotification
ECUM	EcuM_SetWakeupEvent
ECUM	EcuM_CheckWakeup
ECUM	EcuM_EndCheckWakeup

Table 32 Services used by the CANTRCV

7 Configuration

7.1 Configuration with DaVinci Configurator 5

Refer to the integrated online help and parameter descriptions of Configurator 5.

8 AUTOSAR Standard Compliance

8.1 Additions/ Extensions

8.1.1 Memory initialization

To have an independent memory initialization for this BSW module the additional function `CanTrcv_30___Your_Trcv___InitMemory` was added.

8.2 Deviations

While the driver is implemented according [1], some requirements could not be fulfilled in order to ensure proper functionality. The following chapter lists these deviations.

8.2.1 Notification functions

According to SWS [1] the transceiver shall call notification functions in CanIf. As the given CanTrcvChannelId is valid only for one transceiver driver instance, it was decided to call the following notification functions instead:

SWS API	Used API
`CanIf_TrcvModeIndication`	`CanIf_30___Your_Trcv___TrcvModeIndication`
`CanIf_ConfirmPnAvailability`	`CanIf_30___Your_Trcv___ConfirmPnAvailability`
`CanIf_ClearTrcvWufFlagIndication`	`CanIf_30___Your_Trcv___ClearTrcvWufFlagIndication`
`CanIf_CheckTrcvWakeFlagIndication`	`CanIf_30___Your_Trcv___CheckTrcvWakeFlagIndication`

Table 33 Deviation of APIs used by CanTrcv

Within these functions recalculation of the given CanTrcvIndex has to be performed so that the local index of the driver matches the global index of the CanIf.

Affected requirements: CanTrcv086, CanTrcv222, CanTrcv239, CanTrcv238

8.2.2 CanIf_CheckTrcvWakeFlagIndication is always called

According to SWS [1], `CanIf_CheckTrcvWakeFlagIndication` shall be called only if a wakeup was detected. As this would lock CanSm, it was decided to call this notification function in every case, even if no wakeup was detected. If a wakeup was detected, `EcuM_SetWakeupEvent` is called by the transceiver driver.

Affected requirements: CanTrcv238

8.2.3 No re-initialization will occur if POR is set in CanTrcv_SetOpMode

According to SWS [1], the transceiver driver shall perform a re-initialization if POR is set when entering normal mode. This situation can only happen if V_s and V_{cc} drops below the lower boundary and recovers again while MCU is still active. In this case, it is always required to perform a complete re-initialization of the whole MCU as correct functionality

cannot be ensured anymore. During this re-initialization, `CanTrcv_Init` has to be called that reinitializes the underlying transceiver hardware.

Affected requirements: CanTrcv221

8.2.4 Const removed from API CanTrcv_GetTrcvSystemData

According to SWS [1], the API `CanTrcv_GetTrcvSystemData` shall have a parameter `TrcvSysData` of type `const uint32*`. As this parameter has to be filled by the transceiver driver it was decided to remove this `const` qualifier.

Affected requirements: CanTrcv152

8.2.5 Unused BSWMD parameters

According to SWS [1], the parameters `CanTrcvSPICommRetries` and `CanTrcvSPICommTimeout` shall have the multiplicity 1. As the SWS does not describe where to use these values, it was decided to set multiplicity to 0..1 so they do not have to be used.

Affected requirements: CanTrcv172, CanTrcv171

8.2.6 Modified return value of CanTrcv_CheckWakeup

According to SWS [1] the API CanTrcv_CheckWakeup shall return E_OK if a wakeup was detected. As SPI can be asynchronous the return value was modified so that E_OK means that the request to check for Wakeups has been acknowledged. The actual completion of this request may be done in MainFunction.

8.2.7 Initialization of operating mode

According to SWS [1] each CAN transceiver channel shall be initialized to operating mode which is configured by parameter `CanTrcvInitState`. Independent of configuration each CAN transceiver channel is initialized into operating mode NORMAL.

Affected requirements: CanTrcv100, CanTrcv148.

9 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
DEM	Diagnostic Event Manager
DET	Development Error Tracer
ECU	Electronic Control Unit
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
SRS	Software Requirement Specification
SWC	Software Component
SWS	Software Specification
WUP	Wakeup Pattern
WUF	Wakeup Frame
POR	Power-on reset
MCU	Microcontroller Unit
SPI	Serial Peripheral Interface

Table 34 Abbreviations

10 Contact

Visit our website for more information on

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

www.vector.com