Vector CAN Driver

Technical Reference

Reference Implementation 1.5

Version 3.01.01

Authors:	H. Honert, K. Emmert
Version:	3.01.01
Status:	released (in preparation/completed/inspected/released)

1 Document Information

1.1 History

Author	Date	Version	Remarks
Hoffmann	July, 30th 1997	1.00	Initial draft
Baudermann, Ebner	Aug, 9th 1999	2.00	Reorganization of the document Hardware related documentation removed
Ebner	Nov, 2nd 1999	2.01	Spelling corrections
Baudermann	Nov, 6th 1999	2.02	Restrictions with reentrance capability for the following CAN Driver functions: CanInit, CanReset..., CanSleep, CanWakeUp and CAN interrupts
Honert	Dec, 14th 1999	2.03	DLC check added
Ebner	Feb, 8th 2000	2.04	Configuration by tool support (CANgen) added
Baudermann, Rein, Honert, Brändle	May, 23th 2000	2.10	Generally reworked According to reference implementation, version 1.1
Honert	Oct, 31th 2000	2.11	Description of indexed CAN Driver added
Honert	Feb, 28th 2001	2.12	Extensions according to reference implementation version 1.2 Hardware related documentation of HC12 and C16x moved to a separate document
Honert	Aug, 10th 2001	2.13	Description of API extended ■ Single Receive Channel CAN Driver ■ CanCancelTransmit and CanCancelMsgTransmit added ■ Access to ErrorCounters added
Honert, Emmert	Aug, 20th 2001	2.14	Prototype of UserPrecopy corrected Spelling corrections Modifications for pdf conversion
Emmert	Okt, 9th 2001	2.15	Modifications of Figure 4 and 5.
Honert	Mai, 17th 2002	2.16	Function name corrected for indexed driver Extensions according to

			reference implementation version 1.3
Ebner, Honert, Emmert	Jun, 18th, 2003	2.20	Macro names corrected in figure 7. Extensions according to reference implementation version 1.4. Additional explanation for offline / partial offline mode (ch. 5.2.6)
Emmert, Honert	Juli, 29th, 2003	2.21	New tables for API descriptions. Corrections of some Parameters and API descriptions.
Stephan Hoffmann, Klaus Emmert, Heike Honert, Patrick Markl	May 17nd, 2004	2.22	Description of API extended ▪ Direct Transmit Objects Cancel in Hardware Language corrections, New Layout, Technical revisions
Klaus Emmert Matthias Fleischmann	2005-12-30	2.23	GENy added as Generation Tool Added description for: ▪ Multiple ECU ▪ Common CAN ▪ Signal Access Macros ▪ Rx Queue ▪ Conditional Message Received ▪ Variable Datalen
Heike Honert	2006-08-01	2.30	Extensions according to reference implementation 1.5.
Heike Honert	2007-01-09	3.00	prepare links to hw specific Added description for: ▪ CAN RAM check ▪ Standard/HighEnd CAN Driver
Heike Honert	2007-01-29	3.01	some corrections ▪ improve Common CAN ▪ service functions for conditional message reception added ▪ Description for Partial Offline Mode for GENy modified ▪ ESCAN00032527: Update description of ApplCanAddCanInterruptDisable/Restore call-back function
Heike Honert	2010-06-11	3.01.01	Reference to documentation of VstdLib changed

Table 1-1 History of the Document

1.2 Reference Documents

Index and Document Name
[1] TechnicalReference_<hardware>.pdf

Table 1-2 Reference Documents

⚠	**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.

1.3 Contents

Illustrations

Tables

2 About this Document

This document describes the concept, features, API and the configuration of the Vector CAN Driver.

The CAN Driver interface to the CAN Controller is designed to use the hardware specific capabilities in an efficient way. The interface to the higher communication layers is mostly identical for different CAN Controllers, so that the Interaction Layer, Network Management, Transport Protocol and especially the user software are independent of the particular CAN Controller used. Please note that in this document the term Application is not used strictly for the user software but also for all the higher communication layers as listed above. Therefore, Application refers to any of the software modules using the CAN Driver.

Two different types of CAN Driver are supported. These are the Standard CAN Driver and the High End CAN Driver. The High End CAN Driver is an extended Standard CAN Driver. The description of the Standard CAN Driver is also valid for the High End CAN Driver. The additional features of the High End CAN Driver are described in own chapters.

The API of the functions is described in a separate chapter at the end of this document. Referred functions are always shown in the Single receive channel mode.

Hardware related special features and implementation specifics are described in a separate document which is named TechnicalReference_CAN_<hardware>.pdf.

2.1 Documents this one refers to…

- User Manual CAN Driver
- Hardware-specific documentation for the CAN Driver

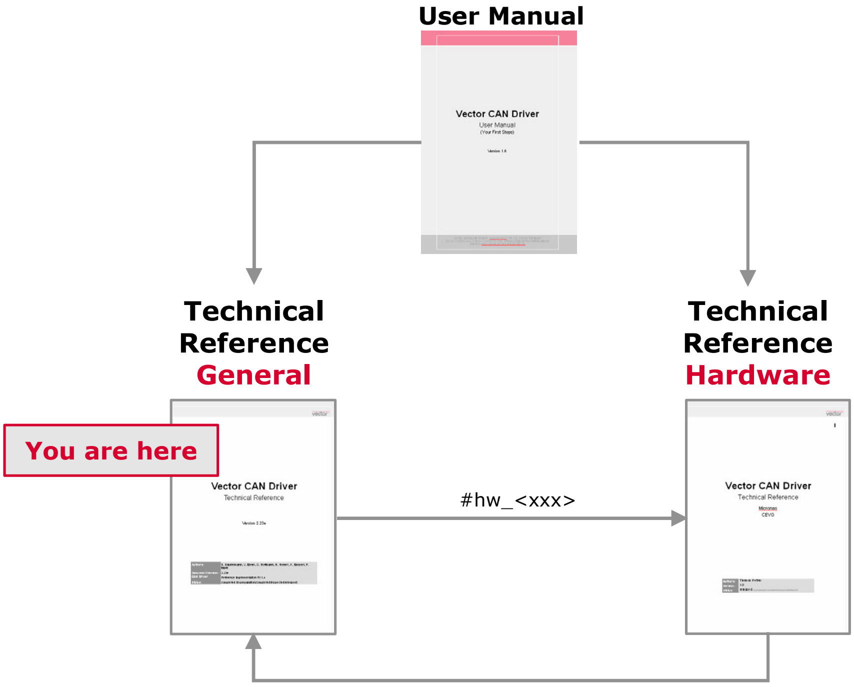


Figure 2-1 Manuals and References for the CAN Driver

2.2 Naming Conventions

Some of the function names are mandatory, because they are used in the CAN Driver. Other names are placeholders, and the Application can redefine or has to select them according to its requirements:

Can... It is mandatory to use all names beginning with Can... as they appear. Can... stands for CAN Driver.

ApplCan... The functions, starting with Appl... are so called callback functions. They are provided by the Application and called by the CAN Driver. They are used to notify the application about events such as state transitions.

User... All names starting with User... are placeholders and will be selected by using the Generation Tool according to the requirements of the Application. User... stands for user-specific functions.

3 Reference Implementations

The reference implementation is a general specification for all Vector CAN Drivers. The software versions for specific CAN Drivers differs, because there are different source codes for different CAN Controllers. Therefore another overall version number exists, representing the reference implementation. The CAN Drivers are implemented according to this reference implementation with an identical feature set and Application interface as well as a harmonized implementation.

3.1 Version 1.0

3.1.1 What's new?

- Identifier ranges defined by acceptance code and mask to receive a complete set of several CAN identifiers. This is much more efficient for special requirements with fixed identifier ranges and can be configured by the Application. Useful settings for Application are selected automatically by the Generation Tool.
- Some parameters are provided by preprocessor defines in the CAN Driver configuration file instead of global variables. This results in more efficient code.
- Notification of a CAN receive message overrun condition is done by the callback function ApplCanOverrun(). This is configurable.
- The internal copy mechanism of the CAN Driver is configurable separately for receive and transmit direction. It will be enabled automatically if an Application data buffer is selected by the Generation Tool.

3.1.2 What's changed?

- General interrupt disable during critical service functions is replaced by a reentrant solution.
- General assertion categories for the following severe errors in the CAN Driver:
  - - User interface (e.g. invalid handles)
  - - Generated data (caused by the Generation Tool)
  - - Hardware problems (unexpected conditions of the CAN Controller)
  - - Internal errors (e.g. inconsistent transmit queue entries)
  - The different categories can be configured separately and the name of the callback function has changed from ApplFatalError(..) to ApplCanFatalError(..).
- Callback function CanMsgReceive() has changed in ApplCanMsgReceived().
- Plausibility check for configuration switches of the CAN Driver is optional and will be done in a separate header file called CAN_CHK.H.
- CanRxActualDLC will be provided as a preprocessor macro.

- If the transmit queue is used for CAN Controllers with several hardware transmit objects, only one of these register sets will be used for normal transmission (in combination with the transmit queue). The others are reserved for Full CAN Transmit Objects with fixed CAN identifier and DLC.
- The names of the following global variables have been changed:
  - CanEcuNumber to canEcuNumber
  - CanRxHandle to canRxHandle

3.2 Version 1.1

3.2.1 What's new?

3.2.1.1 Mandatory (for all CAN Drivers)

- Configurable callback function if software acceptance filtering doesn't match.
- Configurable callback functions to monitor the correct transmit behavior.
- Dynamic transmit objects for variable CAN identifier and DLC
- Security level for the data consistency during the internal copy routines for receive and transmit data.
- Configurable callback functions to control hardware dependent loop break conditions (e.g. during the transition to reset, standby or sleep mode).
- For micros with nested interrupt levels the global disabling of interrupts by CanGlobalInterruptDisable/Restore() is replaced by a configurable interrupt lock level.

3.2.1.2 Optional (for some specific CAN Drivers)

- Support of extended CAN identifiers in different modes (extended only or mixed with standard identifiers)
- Non-interrupt (polling) mode for asynchronous transmission, reception, error and wake-up notification.
- Dynamic transmit objects (for flexible transmit buffer, pretransmit as well as confirmation function).
- Full CAN Transmit Objects with fixed CAN identifier and DLC.
- Dynamic hardware acceptance filtering for the reception of different messages.

3.2.2 What's changed?

- Service functions for flexible CAN identifier and DLC CanTransmitVarDLC/ID(..) must not be used for new developments. It will be replaced by dynamic transmit objects.
- Special macros for the direct access to CAN message information (identifier, DLC, ...) in the receive function (Dir...) will be removed. The standard macros can be used instead.

- Configurable DLC check for the length of the according receive buffer to avoid the overwriting of the next receive buffer: The complete data will not be copied and the Application will not be notified if an inconsistency is detected.
- The return code data type of the CanGetStatus() service function has changed because of additional information in the software state of the CAN Driver and the hardware state of the CAN Controller.

3.3 Version 1.2

3.3.1 What's new?

- Hash search algorithm
- Low level transmit functionality to support e.g. gateways
- Service functions to stop and restart the CAN Controller.
- partial offline to switch dedicated transmit messages off.
- New return code of CanTransmit() in case of partial offline.
- Macros which return 8 bit of a received extended ID for use in precopy functions.
- Access to error counter of the CAN controller
- Service function to cancel transmit requests and confirmations

3.3.2 What's changed?

- Generic Precopy function is now mandatory
- CanSleep() and CanWakeUp() has now a return value.
- CanGetStatus() is always available. Activation of extended status enables the additional information in the hardware state of the CAN Controller.
- Passive mode can only be activated for all transmit requests and not for dedicated messages.
- The name of some macros to access the ID in a precopy function has changed
- In the indexed CAN Driver, CanGetDynTxObj() has the channel as additional parameter.
- Macro CanRxActualIdHi renamed to CanRxActualIdRawHi
- Macro CanRxActualIdLo renamed to CanRxActualIdRawLo

3.4 Version 1.3

3.4.1 What's new?

- New service functions to disable and restore CAN interrupts

3.4.2 What's changed?

- Function CanInterruptDisable renamed to CanGlobalInterruptDisable

- Function CanInterruptRestore renamed to CanGlobalInterruptRestore
- Support of systems with mixed Identifier expanded
- Macro CanRxActualId returns the Identifier always in the logical presentation

3.5 Version 1.4

3.5.1 What's new?

3.5.1.1 Mandatory (for all CAN Drivers)

3.5.1.1.1 Common features

- New functions CanCopyFromCan and CanCopyToCan. Hardware/Compiler dependent functions to optimize copying of data (provide for higher layers such as TP, Diag). more… API…
- The CAN driver can be configured to run without any disabling of interrupts. The application has to take care of reentrancy! To set the Can Driver to this mode, the security level has to be set to the lowest value. more…
- The CAN Driver is more fault tolerant against unexpected CAN interrupts like Rx interrupt of a transmit object. Interrupt in polling mode or interrupts of unused objects are handled by the driver.
- Callback function ApplCanPreWakeUp which is called immediately after the activation of the wakeup interrupt. more... API…
- The CAN Driver doesn't use library function of the compiler library (except for intrinsic functions)
- The Can Driver code is MISRA compliant.

3.5.1.1.2 Transmission features

A confirmation function common to all transmit messages is supported. This function is called after any successful transmission (except Direct Transmit Objects but includes canceled transmit objects that had been sent although). more… API…

3.5.1.2 Optional (for some specific CAN Drivers)

3.5.1.2.1 Transmission features

- CanDirectTransmit(txHandle) to support direct transmit objects.

This transmission is completely independent of other transmit messages and can be sent e.g. out of a NMI (non-maskable interrupt service routine. (see also what's changed).

3.5.1.2.2 Reception features

- For Full CAN controllers polling of Basic CAN is supported (all functionality of the CAN driver can be used in polling mode). more…
- A callback function is called, if the DLC check fails (this means if the DLC of the received message is shorter than configured for this message). more… API…

- Support variable data length (for specific OEMs).

3.5.2 What's changed?

- The names of the different kinds of transmission objects changed. To make the differences clear in the following table all kinds of transmission objects are listed even if nothing changed. more…

Old names (before RI 1.4)	New names (RI 1.4 and later)
Transmit Objects	Normal Transmit Object
Direct Transmit Objects	Full CAN Transmit Object
Dynamic Transmit Objects	Dynamic Transmit Objects
---	Direct Transmit Objects
Low Level Message Transmit	Low Level Message Transmit

3.5.2.1 Transmission features

- The interface of the TxObserve Callback functions has changed (parameter of the functions ApplCanTxObjStart() and ApplCanTxObjConfirmed() and ApplCanInit(). An additional parameter is used. This additional parameter is the handle of the hardware object (a unique number over all hardware transmit objects which starts with 0). more… API…
- The functions CanCancelTransmit() and CanCancelMsgTransmit can now delete a message in the hardware transmit buffer as well as in the queue. more… API…
- To get the tx handle of a pending transmit message, a new Service function is defined: CanTxGetActHandle(CanObjectHandle logTxHwObject) more… API…
- If a CAN controller doesn't support arbitration by ID, Direct Transmit Objects and Full CAN Transmit Objects have a higher priority than the Normal Transmit Object. The Low Level Message transmission has the lowest priority. more…
- It is not possible/necessary any longer to specify the number of the CAN transmit buffer for Full CAN Transmit Objects. This will be done by the Generation Tool automatically.
- The functions CanPartOffline and CanPartOnline are designed to be reentrant. API…

3.6 Version 1.5

3.6.1 What's new?

- data size optimized Tx Queue.
- In systems with mixed IDs (standard and extended), each range can be configured to standard or extended ID individually.
- Each hardware objects can be configured individually to polling or interrupt mode. more... API...
- Multiple Basic CAN objects can be defined to optimize the hardware filters. more...

- Rx Queue supports now queuing of messages out of a range.
- Notification about mode change of the CAN driver between offline and online mode.
- New macros to fill structure of Low Level Message Transmit
- distinguish between Standard CAN Driver and High End CAN Driver instead of optional features

3.6.2 What's changed?

- Source Address of Range Specific Precopy Messages removed – deviation to HIS CAN Driver Specification. (EcuNumber isn't any longer a member of tCanRxInfoStruct)
- Return code of Range Precopy Functions has effect on further reception handling.
- Direct Transmit Objects are not supported any more
- API Categories Single Receive Buffer (SRD) and Multiple Receive Buffer (MRB) are not supported any more.
- Global Interrupt Control has been moved to VStdLib (CanGlobalInterruptDisable(),. CanGlobalInterruptRestore(), Interrupt Control by Application, Interrupt Lock Level). ...more information see Application Note AN-ISC-2-1050_VstdLibIntegration.pdf.
- Channel parameter for Hardware Loop Check Callbacks – deviation to HIS CAN Driver Specification API...
- The following macros are not available any more: `MK_EXTID_LO`, `MK_EXTID_HI`, `MK_STDID_LO`, `MK_STDID_HI`, `CanRxActualIdRaw`, `CanRxActualIdRawHi`, `CanRxActualIdRawLo`
- The polling Tasks are allowed to be called in Sleep mode, too.
- Improvement of usage of assertions
- Same OSEK OS interrupt category for all CAN interrupts.
- Variable data length replaced by copying data with received DLC and DLC check against minimum length.
- Description of dynamic pretransmit function, dynamic confirmation function and dynamic acceptance filtering removed.

4 Overview

For error prevention, maintainability and expandability of Application programs, it is essential to have a uniform interface between Application and CAN Driver, mostly independent of the CAN Controller used. The CAN Driver itself must be adapted to the CAN Controller for reasons of efficiency. This yields the following requirements for a universally applicable CAN Driver:

- Independent of Application
- Driver code optimized for the CAN Controller used
- Uniform interface to the Application for different CAN Controllers
- Efficient usage of hardware resources, especially RAM and run time
- Support of special services like Interaction Layer, Network Management, Transport Protocol

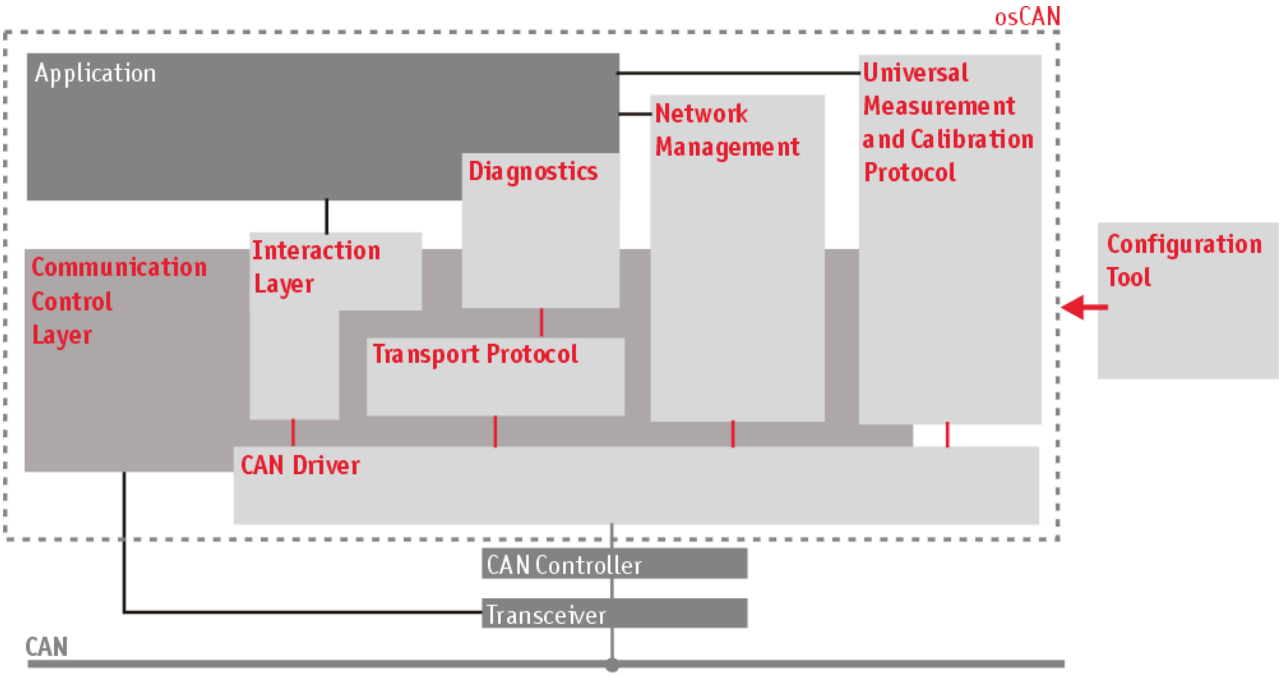


Figure 4-1 Relationship of the individual Software Components. They are customized by the Generation Tool

The generic CAN Driver code is independent of the Application. Only the callback functions have to be given by the Application. The Application specific description data are stored in dedicated data structures. The structure of the description data is fixed; however, the contents of the structures are defined according to the ECU specific behavior by the Generation Tool. This is done partly automatically based on information in the CAN database and partly manually by user specific settings in the Generation Tool. The data structures are specific for the CAN Controller, they are linked to the CAN Driver code (ROM-capable).

The data to be transmitted or received are exchanged by default via global data buffers. These data buffers are CAN message based. They are also created by the Generation

Tool. Additionally, the Generation Tool creates signal-based access macros and/or functions. This means the Application does not have to know the location and the structure of the global data buffers. The names of the access macros/functions are formed from the signal names in the CAN database. The detailed structure and features of the access macros/functions are described in the documentation of the Generation Tool.

The Generation Tool will not be discussed in this CAN Driver documentation, since it is irrelevant for the CAN Driver functionality. But the generated data structures are highly optimized for an efficient usage of each CAN Controller. Therefore the usage of the Generation Tool is a must to customize the CAN Driver code to the special needs of the Application.

4.1 Short Summary of the Functional Scope

In this section the main tasks of the CAN Driver are summarized very briefly:

1. Initialize the CAN Controller
2. Transmit a single CAN message
3. Receive a single CAN messages
4. Handle Bus-Off
5. Support sleep mode
6. Support special services

4.1.1 Initialization

There are several CAN Driver service functions for initialization purposes available. CanInitPowerOn(..) for the complete initialization of software and hardware after power-on, CanResetBusOffStart(..) and CanResetBusOffEnd(..) for the re-initialization of the CAN Controller after BusOff. For any other re-initialization the application can call CanInit(..).

Various initialization data structures can be predefined by the Generation Tool and referenced in the Application by means of a specific initialization handle.

4.1.2 Transmission

One of the main services provided by the CAN Driver is to set up a transmit request in the CAN Controller by the service function CanTransmit(..). The reference to the CAN message specific description data is done by a transmit handle used in the Application. This information like CAN identifier and DLC is set up by the Generation Tool based on the CAN Database and additional user specific settings. If the CAN Controller is busy because all hardware transmit objects are currently reserved, the transmit request can be stored temporarily in a transmit queue. The number of hardware transmit objects depends on the CAN Controller or CAN Driver configuration. For details please refer to the CAN Controller specific documentation TechnicalReference_CAN_<hardware>.pdf [#hw_comObj]. If the CAN Controller is ready, data can be copied by different mechanism to the hardware transmit registers. The return code of the transmit function informs whether the transmit request was accepted by the CAN Driver or not. If it was rejected by an error and no transmit queue is used, the Application is responsible for the repetition of the transmit request until the message is in the CAN Controller. A successful transmission is signaled by a confirmation. Special features like offline or passive mode are available to control the transmit path of the CAN Driver by the Network Management or Diagnostics.

4.1.3 Reception

If a message on the CAN bus was accepted by the hardware and software acceptance filtering, data can be read by different mechanism and this asynchronous event is notified to the application by an indication. Additionally a special callback function ApplCanMsgReceived() allows the user to access receive data directly in the scope of a receive interrupt before the software acceptance filtering. The algorithm for the software acceptance filtering is configurable because in some Applications a lot of irrelevant CAN identifiers are passing the hardware acceptance filter and an efficient software filtering is very important.

4.1.4 Bus-Off

The CAN Driver notifies a detected BusOff state to the Application by calling a special callback function ApplCanBusOff(). Further processing like re-initialization of the CAN Controller and additional customer-specific requirements like disabling transmissions for a certain time have to be done by the Application.

4.1.5 Sleep Mode

Some CAN Controller are supporting a so called sleep mode with reduced power consumption. The CAN Driver provides the service functions CanSleep() and CanWakeUp() to enter and leave this special mode on request of the Application.

If the CAN Controller is also wakeable by the CAN bus, the callback function ApplCanWakeUp() is called if this condition is detected. In some cases this leads to the situation that the CAN controller is initialized (CanWakeUp) before the application will be notified.

In case of changing the PLL (SLEEP = slow speed / ACTIVE = normal speed) the application must be informed immediately. Otherwise the “long” interrupt execution causes a watchdog reset. Therefore the callback function ApplCanPreWakeUp is called just after the activation of the wakeup interrupt. The configuration is done via the Generation Tool or user configuration file.

4.1.6 Special Features

There is additional support for special features like

- Status of CAN Driver and CAN Controller
- Interrupt control
- Assertions
- Hardware loop check
- Support of OSEK compliant operating systems
- Multiple-channel CAN Driver
- Polling mode

- Handling of Extended Identifiers
- Stop mode

4.2 Data Structures for CAN Driver Customization

The description data created by the Generation Tool are split into initialization structures for the CAN Controller as well as transmission and reception structures for CAN messages. They are located in the ROM memory of the microprocessor. The receive and transmit buffers are mapped in RAM data and will be referenced by the description data. The description data also contains references (pointers) to user-specific functions of the Application. The CAN Driver accesses all the structures in the description data. The CAN Driver is independent of the Application but the generated description data depends on the particular Application.

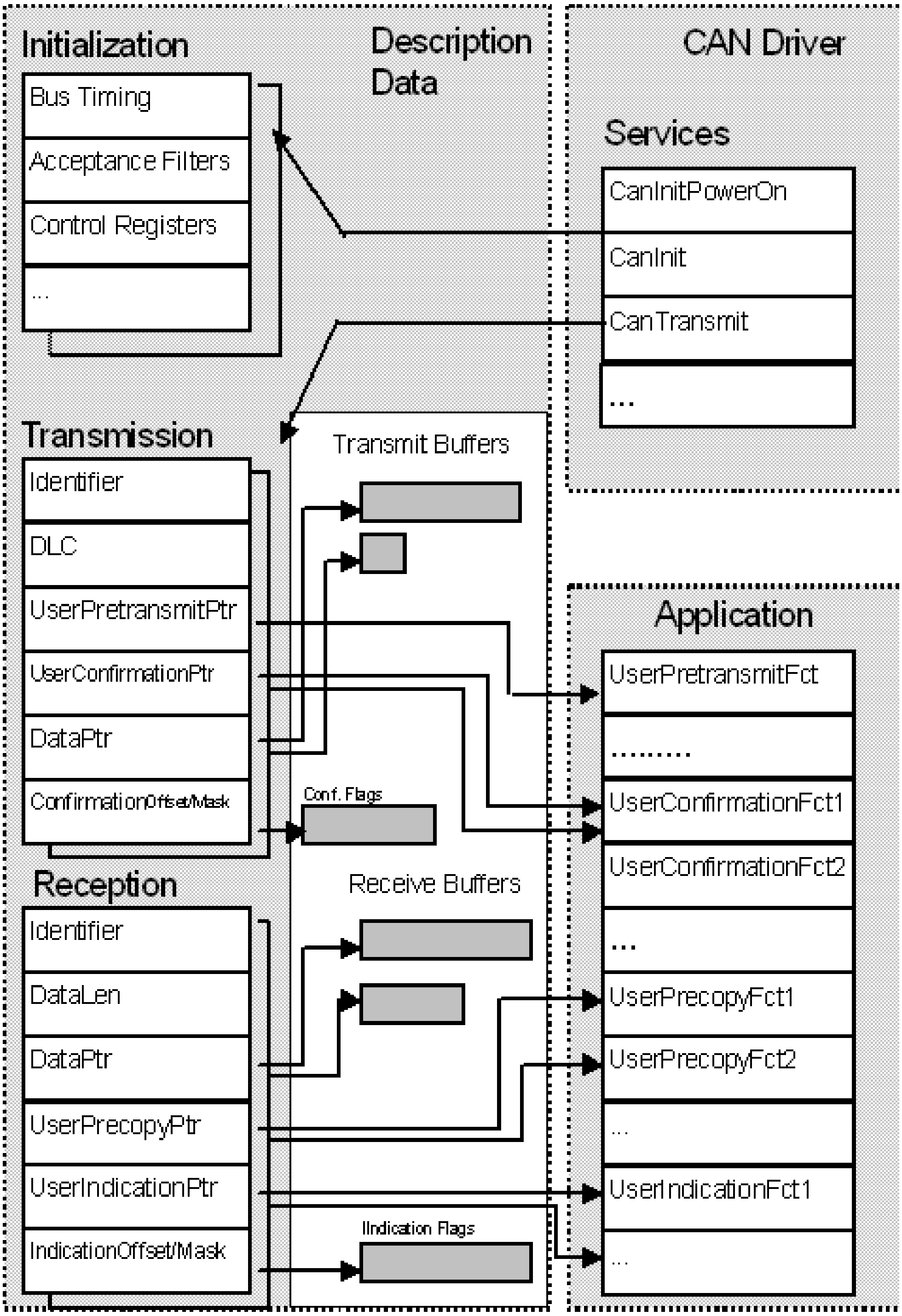


Figure 4-2 Description data, CAN Driver and Application with their interfaces.

4.2.1 ROM Data

4.2.1.1 Initialization Structures

The CAN Controller is initialized with the description data stored in the initialization structures. They consist of the register values for the CAN Controller. They are highly dependent on the particular used CAN Controller.

4.2.1.2 Transmit Structures

The structures listed below are used by the CAN Driver internally.

ID	Identifier of the messages to be transmitted. The format is CAN Controller dependent for efficiency reasons.
DLC	Number of data bytes to be transmitted (Data Length Code). The format is CAN Controller dependent for efficiency reasons.
DataPtr	Pointer to the CAN message based global transmit buffer.
UserPreTransmitPtr	Pointer to the user specific pretransmit function (must be a NULL pointer if not used).
UserConfirmationPtr	Pointer to the user specific confirmation function (must be a NULL pointer if not used).
ConfirmationOffset/Mask	Byte offset and bit mask for the CAN Driver access to the corresponding confirmation flag.

4.2.1.3 Receive Structures

The structures listed below are used by the CAN Driver internally.

ID	Identifier of the messages to be received. The format is CAN Controller dependent for efficiency reasons.
DataLen	Number of data bytes to be copied. The value may be different from the DLC of the message received. The driver then only copies the number of bytes stored in this structure. The other bytes are ignored.
DataPtr	Pointer to the CAN message based global receive buffer.
UserPrecopyPtr	Pointer to the user specific precopy function (must be a NULL pointer if not used).
UserIndicationPtr	Pointer to the user specific indication function (must be a NULL pointer if not used).
IndicationOffset/Mask	Byte offset and bit mask for the CAN Driver access to the corresponding indication flag.

4.2.2 RAM Data

The RAM data consist of transmit and receive buffers for the CAN messages.

In the data buffers, the first byte transmitted or received is located at the least significant address of the data array (Note: Bit 7 is transmitted first).

For some microprocessors there are memory areas which can be accessed more efficiently (e.g. internal RAM or bit addressable segments). The data buffers can be mapped by the Generation Tool.

5 Detailed Description of the Functional Scope (Standard)

5.1 Initialization

5.1.1 Power-On Initialization of the CAN Driver

The following service function must be called once after power-on to initialize the CAN Driver:

```
void CanInitPowerOn( void );
```

This call initializes the CAN Controller for each channel and all CAN Driver variables (local and global), i.e. the CAN Driver is set to online and active state.

This service function has to be called for a proper initialization before any other CAN Driver function and before the global interrupts are enabled.

5.1.2 Re-Initialization of the CAN Controller

The CAN Controller is completely re-initialized by the service function call:

```
void CanInit( CanInitHandle initObject );
```

The parameter initObject means a handle for a specific initialization structure.

It is a must to bring the CAN Driver into offline state before this service function is called.

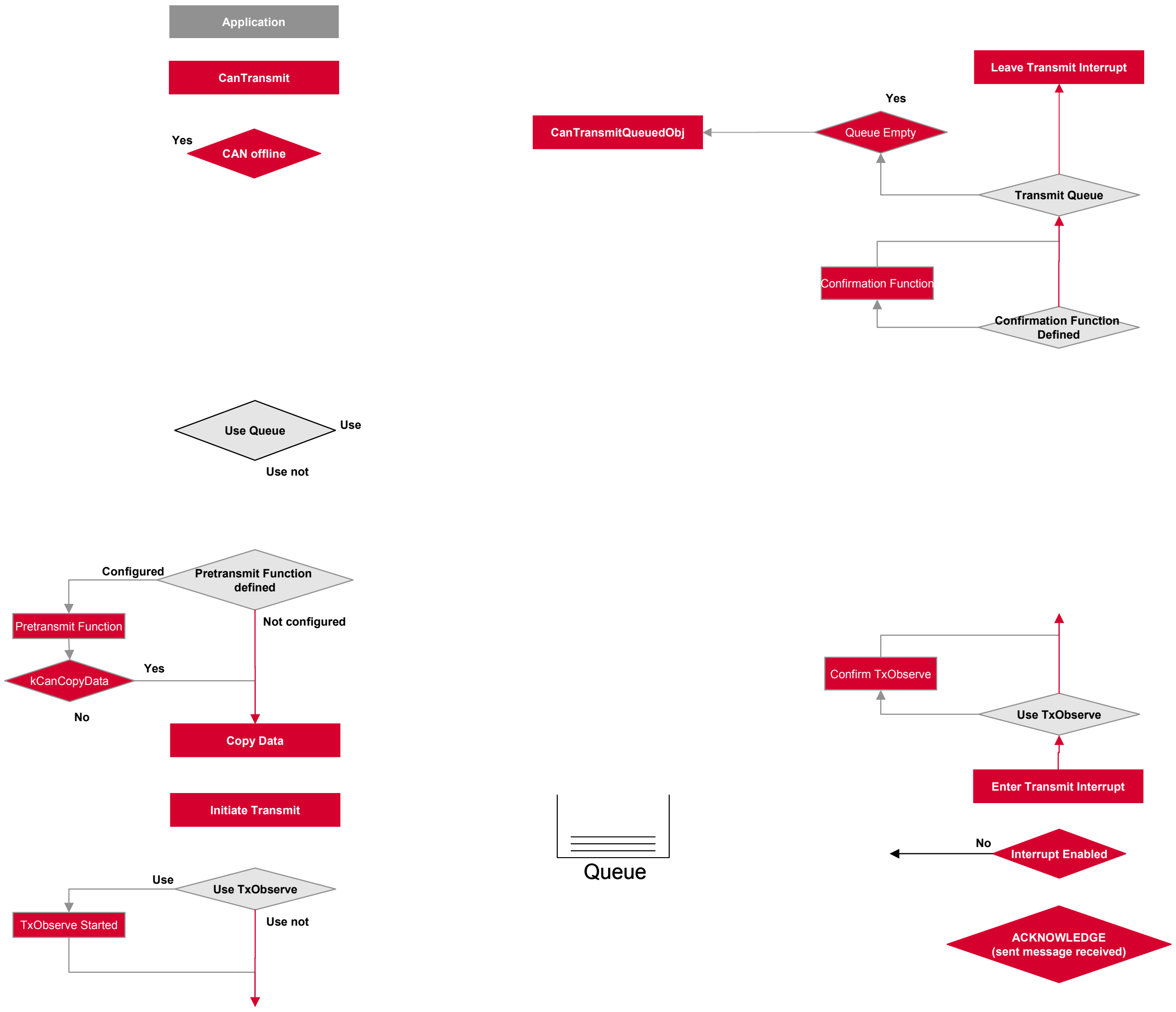
By this service function only the CAN Controller and the corresponding internal variables will be initialized. Software states like online/offline or active/passive remain unchanged.

Changes of individual registers of the CAN Controller are only possible by means of a complete re-initialization, i.e. an entire initialization structure must be provided for each register change (e.g. bit timing, acceptance filtering, ..).

5.2 Transmission

5.2.1 Detailed Functional Description

This section shows the transmission of a CAN message using different methods. For the general processing first a flow chart is used. The gray decision symbols branch to features that can be removed from the CAN Driver using the configuration options (see Figure 5-1). In a second step sequence charts are used to show how the different objects of the CAN Driver, description data and Application program work together.


Application
CanTransmit
Yes
CAN offline
Use Queue
Use
Use not
Configured
Pretransmit Function defined
Pretransmit Function
Not configured
Yes
kCanCopyData
No
Copy Data
Initiate Transmit
Use
Use TxObserve
TxObserve Started
Use not
CanTransmitQueuedObj
Yes
Queue Empty
Leave Transmit Interrupt
Transmit Queue
Confirmation Function
Confirmation Function Defined
Queue
Confirm TxObserve
Use TxObserve
Enter Transmit Interrupt
No
Interrupt Enabled
ACKNOWLEDGE
(sent message received)

The main service function to initiate a transmit request is

```
vuint8 CanTransmit( CanTransmitHandle txObject );
```

The function parameter is a transmit message handle. It represents an index in the generated transmit description data. The return code contains the following information:

`kCanTxOk`	Successful transmit request. The message is sent out by the CAN Controller without any further action required. For CAN Drivers with transmit queue, this return code is also used if the transmit request has been accepted in the queue, even if it was already in queue.
`kCanTxFailed`	CAN transmit request failed. In this case the calling application has to repeat the transmit request later.
`kCanTxPartOffline`	Error code because CAN Driver's transmit path is in partial offline mode for this transmit object.

The left path (see Figure 5-1) time flows from top to bottom. This path shows the program flow calling the service function CanTransmit(..). First the CAN Driver checks whether the transmit path is switched to offline state. If so the function returns with an error code. Then the Driver checks (if configured) the partial offline mode. If the specified message is offline, the function will return an error code.

In the next step the CAN Driver checks the availability of a hardware transmit object. If no object is available the transmit request is stored in the transmit queue (if configured to be used) and the CAN Driver returns to the Application with the return code kCanTxOk. If no transmit queue is used the CAN Driver returns with an error code kCanTxFailed.

If a transmit object is available the CAN identifier and the data length code will be set in accordance to the description data. Now, if a pretransmit function is configured, this pretransmit function will be called. Within this user specific function the Application may copy the data to be transmitted directly to the CAN Controller hardware registers. If the data is completely copied, the pretransmit function returns kCanNoCopyData to the CAN Driver.

The data has to be copied by the CAN Driver itself, if there is no pretransmit function defined or this function returns kCanCopyData. In this case the CAN Driver copies the data from the global data buffer associated with the message to the CAN Controller hardware registers.

Then the transmission of the CAN message is started in the CAN Controller and the function returns the code kCanTxOk to the Application.

Dependent on the configuration, the TxObserve function is now started.

In the right path of the figure below, the time flows from bottom to top. This path shows the program flow in the interrupt service routine after a successfully transmission of the message to the CAN bus. In the transmit interrupt routine, the confirmation actions are performed. If configured, first the TxObservation is confirmed, then (if configured) a confirmation function for all messages is called. Afterwards the confirmation flag is set and then the message-specific confirmation function is called.

If the CAN Driver is configured to use a transmit queue, after processing the confirmation actions the CAN Driver checks if the transmit queue is empty. If so the transmit interrupt routine is finished. If there are entries in the queue the highest priority CAN message is removed from the queue and the transmission of this message is requested. This is also done on interrupt level.

In the middle of the picture we see the transmit queue which is used if all hardware transmit objects are busy, when CanTransmit(..) is called.

The next sections describe the transmission of a CAN message using sequence charts. The vertical lines within these diagrams represent program objects like interrupt routines, functions (thick lines) or data objects (thin lines). The horizontal lines represent program flow or data access within the program. Flow control and program instances are described using thick lines, data access is described using thin lines. Time flows from the top of a chart downwards so that sequence „1“ is performed before sequence „2“. The description of the sequence charts is given in the tables following the charts.

The first sequence chart in Figure 5-2 shows the behavior if a hardware transmit object is available, a global data buffer is associated to the message and the copy mechanism of the CAN Driver is used.

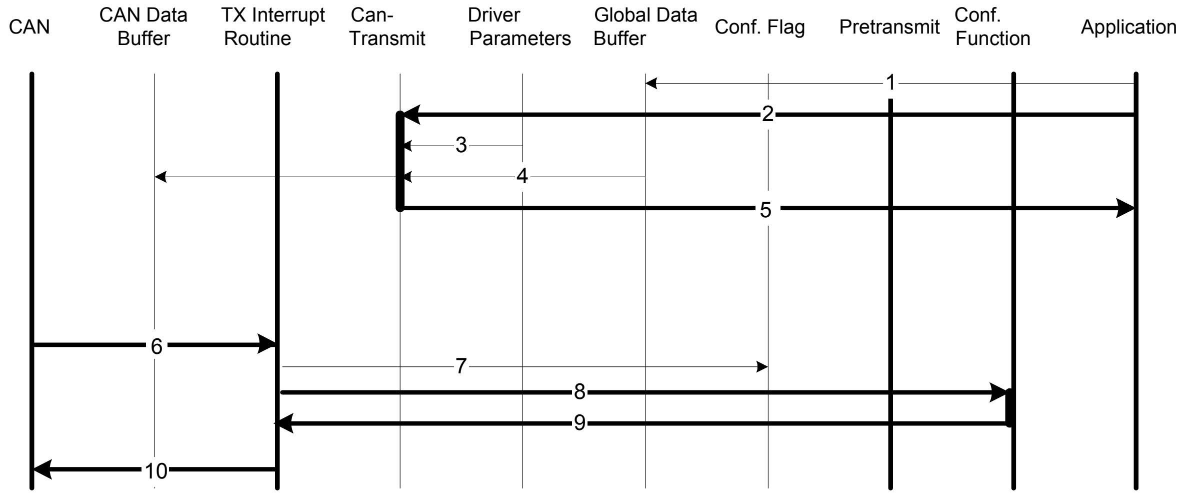


Figure 5-2 Transmission with an available transmit object; Using global data buffer

No	Description
1	The Application writes the data to the global data buffer
2	The Application calls CanTransmit(..) service function
3	Function uses description data (CAN identifier, DLC, etc...)
4	Global data buffer is read and copied; the transmit process is started
5	CanTransmit(..) service function is finished, the return code is kCanTxOk
6	The message is successfully sent to the CAN bus. Transmit interrupt routine is started
7	Transmit confirmation flag is set (cleared by the Application)

8	Confirmation function is called
9	Confirmation functions returns to transmit interrupt routine
10	Transmit interrupt routine is left

The next sequence chart in shows the behavior if a hardware transmit object is available and a pretransmit function is used to copy the data to be sent.

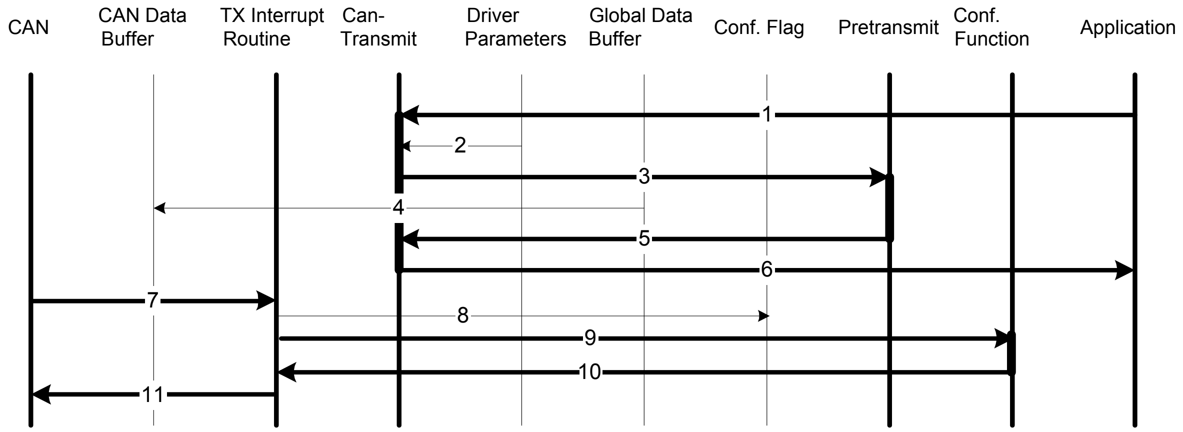


Figure 5-3 Transmission with an available hardware transmit object; Using a pretransmit function to copy data

No	Description
1	CanTransmit(..) service function is called by the Application
2	Function reads the description data (CAN identifier, DLC, etc.)
3	Call of the pretransmit function
4	Pretransmit function writes data to the CAN Controller
5	Pretransmit function returns to CanTransmit(..)
6	Start transmission; CanTransmit(..) service function is finished and the return code is kCanTxOk
7	The message is successfully sent to the CAN bus. Transmit interrupt routine is started
8	Transmit confirmation flag is set (cleared by the Application)
9	Confirmation function is called
10	Confirmation function returns to transmit interrupt routine
11	Transmit interrupt routine is left

The next sequence chart in shows the behavior, if no hardware transmit object is available. This sequence chart is valid only if the CAN Driver is configured to use a transmit queue. The data is copied by the CAN Driver itself.

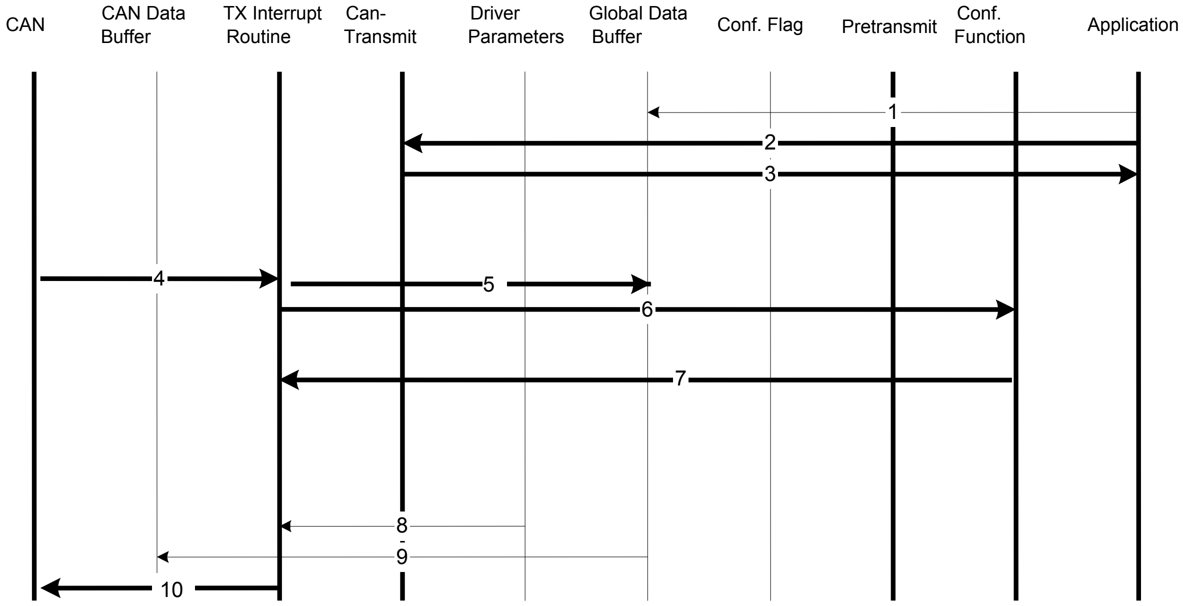


Figure 5-4 Transmit procedure if no hardware transmit object available

No	Description
1	The Application writes the data to the global data buffer
2	The Application calls CanTransmit() service function. No hardware transmit objects available. Request is stored in the transmit queue.
3	Function returns kCanTxOk
4	Transmit interrupt: A (previous) CAN message was successfully sent, transmit object is available again
5	Confirmation flag of the previous CAN message is set (cleared by the Application)
6	Confirmation function of the previous CAN message is called
7	Confirmation function return
8	The transmit queue is checked for requests. The pending transmit request is found. The description data are evaluated (CAN identifier, DLC, etc...)
9	Global data buffer is read and copied; the transmit process is started
10	Transmit interrupt routine is left

5.2.2 Transmit Queue

The normal Tx object can be configured to use a transmit queue or not. The Transmit Queue is not available for Full CAN Objects and the Low Level Transmit Object. If no transmit queue is used, the Application is responsible to restart a transmit request if it wasn't accepted by the CAN Driver. In case of using a transmit queue, a transmit request is always accepted (if the driver is online). But the transmit queue holds only the transmit request of a CAN message. It doesn't store the data to be sent. Please note the same message can be queued only once. The CAN Driver sets a transmit request in the transmit queue, if no hardware transmit object is available after CanTransmit(..) is called. On a transmit interrupt, i. e. when a message has been sent successfully, the CAN Driver checks whether transmit requests are stored in the queue. If so, one requests is removed from the queue and the transmit request is executed. The search algorithm in the queue is priority based, there is no FIFO strategy. This means the CAN identifier with the lowest number is removed first from the queue.

If the CAN Driver is configured to use a transmit queue, the internal data copy mechanism will be initiated and/or the pretransmit function will be called in the scope of a transmit interrupt after the completion of a previous transmit request. Therefore the user has to guarantee the data consistency, because an Application write access to the global data buffer may be interrupted by such a transmit interrupt. If within this interrupt the associated message is requested to be transmitted on the CAN bus, inconsistent data may be sent. The Application must ensure data consistency by one of the following mechanisms:

- Disable Interrupts while writing data to the global data buffer
- Use the message based confirmation flag to manage the data access handling. On startup the access right is on Application side. Calling CanTransmit(..) this access right is given to the CAN Driver. As soon as the confirmation flag is set by the CAN Driver, the access right is given back to the Application.
- In polling mode the service function CanTxTask() must be used to transmit queued messages. The transmission of a CAN message is only started if the CanTxTask() is called. In polling mode every message is queued in the transmit queue. To ensure that every message was send the CanTxTask() may be called cyclic.

5.2.3 Data Copy Mechanisms

There are two different methods for the Application to pass the data to be transmitted to the CAN Driver. The CAN Driver selects the method for each message depending on the CAN Driver description data. If no pretransmit function is defined, the usage of a global data buffer is a prerequisite and the CAN Drivers internal data copy mechanism is always used. If a pretransmit function is defined, the data to be transmitted may be stored anywhere in the Applications memory and the user defined copy mechanism in the pretransmit function is used.

5.2.3.1 Internal

With the internal data copy mechanism, the Application writes the data to be transmitted to a global data buffer associated with the transmit message. The global data buffer is defined by the Generation Tool. The access to the global data buffer is done by means of access macros and/or functions which are also defined by the Generation Tool. After passing the data to the global data buffer, the Application initiates the transmit request by calling CanTransmit(..) and the data is copied internally to the CAN Controller hardware registers.

Important
Data consistency of CAN messages has to be guaranteed by the Application if CanTransmit(..) is called on a higher interrupt or task level, or the transmit queue is used.

5.2.3.2 User defined (“Pretransmit Function”)

Using the pretransmit function to pass the data to be transmitted to the CAN bus, the Application first initiates the transmit request by calling CanTransmit(..). Just before the message is put in the CAN chip, the CAN Driver calls a user defined pretransmit function. For each transmit message a separate pretransmit function may be defined. Within this user specific function the user can write the data directly to the hardware registers of the CAN Controller, but other tasks can also be performed. The return code of the pretransmit function indicates to the CAN Driver whether the data are to be copied by the CAN Driver internally from the global data buffer to the CAN Controller hardware registers or not (if it is already done within the pretransmit function).

Important
Be careful if a pretransmit function is used. Interrupts are not disabled during the call of this user specific function by the CAN Driver, therefore the restrictions for security level 0 are valid. If the interrupts are not disabled before and restored after the copy process by the Application, data consistency of a CAN messages cannot be guaranteed if the transmit queue is used.

5.2.4 Notification

After the successful transmission of a message on the CAN bus (i.e. at least one other CAN bus node received the CAN message correctly with an acknowledge), the Application can be notified by different confirmation mechanisms:

5.2.4.1 Data Interface (Confirmation Flag)

If a confirmation flag is used, this message related flag is set by the CAN Driver, if the associated CAN message was sent on the CAN bus. This is done in the scope of the transmit interrupt. The flag must be cleared by the Application.

Important
Interrupts have to be disabled while the confirmation flags are being cleared, because of the read-modify-write conflict if this operation is interrupted by a CAN transmit interrupt routine. This can result in the loss of events.

5.2.4.2 Functional Interface (Confirmation Function for each message)

In parallel or instead of the data interface a functional interface can be configured, i.e. user specific function is called if the associated CAN message was sent on the CAN bus. This is also done in the scope of the transmit interrupt and therefore special care of the run time of this function has to be taken.

5.2.4.3 Functional Interface (Common Confirmation Function for all messages)

A common confirmation function informs the application via ApplCanTxConfirmation about a successful transmission of a message. Any message is confirmed via this callback function.

Info
A canceled transmission will provoke a notification if the message was send on the bus. If the message had been deleted out of the hardware, the application will not be notified.
More…

5.2.5 Offline Mode

The CAN Driver's transmit path can be switched to the offline state, i.e. disabled. In this state no CAN messages are sent to the CAN bus. On each transmit request the CAN Driver checks the internal flag which indicates whether the transmission is currently disabled and the transmit service function returns an error code. This flag is set and reset by the following CAN Driver service functions

```
void CanOnline( void );

void CanOffline( void );
```

These CAN Driver service functions are called by the Network Management or by the Application (only if there is no Network Management available on a specific CAN channel).

The Application can be notified about the mode change (e.g. if the Network Management calls `CanOnline()` or `CanOffline()`). This is done with the following callback functions:

```
void ApplCanOnline( void );

void ApplCanOffline( void );
```

5.2.6 Partial Offline Mode

The partial Offline Mode enables the application to prevent the transmission of groups of CAN messages. CanTransmit() returns a special code, if the requested message cannot be sent because of the active partial offline mode. The partial offline mode is implemented by the following functions:

```
void   CanPartOnline ( vuint8 sendGroup );

void   CanPartOffline( vuint8 sendGroup );

vuint8 CanGetPartMode( void );
```

The partial offline mode can handle up to eight different groups of messages. The function parameter sendGroup decides about this group. CanPartOffline() switches all messages of one ore more send groups to the offline state. Earlier calls of CanPartOffline() are not affected. CanPartOnline() switches one or more send groups back to online state.

Each message might be assigned to one or more send groups. The names of the send groups are configurable. Each send group can be switched to offline or online by using the generated define:

```
C_SEND_GRP_<name>
```

C_SEND_GRP_ALL can be used to switch all groups together to offline or online.

Example

The following table shows, which message is assigned to which send group (CANgen concept. For GENy concept, go to the next chapter.).

	send group name							
	7	6	5	4	3	User2	User1	User0

MESSAGE1					x	x	x	
MESSAGE2						x		x
MESSAGE3					x	x		

Example
for a possible program flow:

CanPartOffline(C_SEND_GRP_User0); MESSAGE2 is stopped to be send

CanPartOffline(C_SEND_GRP_User1); MESSAGE1 is stopped to be send
MESSAGE2 is still stopped to be send

status = CanGetPartMode(); status is equal to (C_SEND_GRP_User0
| C_SEND_GRP_User1)

CanPartOnline(C_SEND_GRP_User0); MESSAGE1 is still stopped to be sent
MESSAGE2 can be sent again

CanPartOffline(C_SEND_GRP_User0 | C_SEND_GRP_3);
MESSAGE1 is stopped to be sent
MESSAGE2 is stopped to be sent
MESSAGE3 is stopped to be sent

CanPartOnline(C_SEND_GRP_ALL); All send groups are online again. All messages can be sent now.

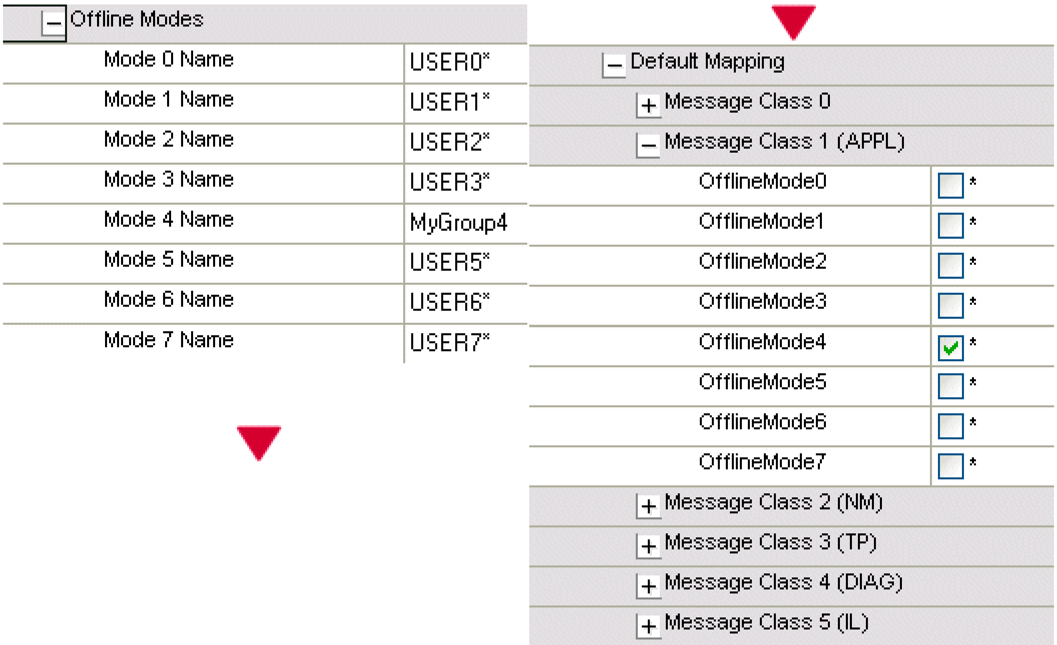
Info
If the offline mode and partial offline mode are used in parallel the offline mode has ‘higher priority’. This means if the offline mode is set the function CanTransmit always returns ‘kCanTxFailed’ independent of the current partial offline state.

5.2.6.1 Partial Offline Mode with GENy

In GENy there are

- 8 Offline Modes (SendGroups)
  - Default name is UserX, but can be changed as shown in the illustration below. There **Offline mode 4** is changed to **MyGroup4**.
- 5 Message Classes for
  - Default (0)
  - Appl
  - Nm
  - Tp
  - Diag
  - Il

All messages are assigned automatically to a message class using their attribute information from the DBC file.



⊟ Offline Modes

Mode 0 Name	USER0*
Mode 1 Name	USER1*
Mode 2 Name	USER2*
Mode 3 Name	USER3*
Mode 4 Name	MyGroup4
Mode 5 Name	USER5*
Mode 6 Name	USER6*
Mode 7 Name	USER7*

⊟ Default Mapping

- ⊞ Message Class 0
- ⊟ Message Class 1 (APPL)

OfflineMode0	□ *
OfflineMode1	□ *
OfflineMode2	□ *
OfflineMode3	□ *
OfflineMode4	☑ *
OfflineMode5	□ *
OfflineMode6	□ *
OfflineMode7	□ *

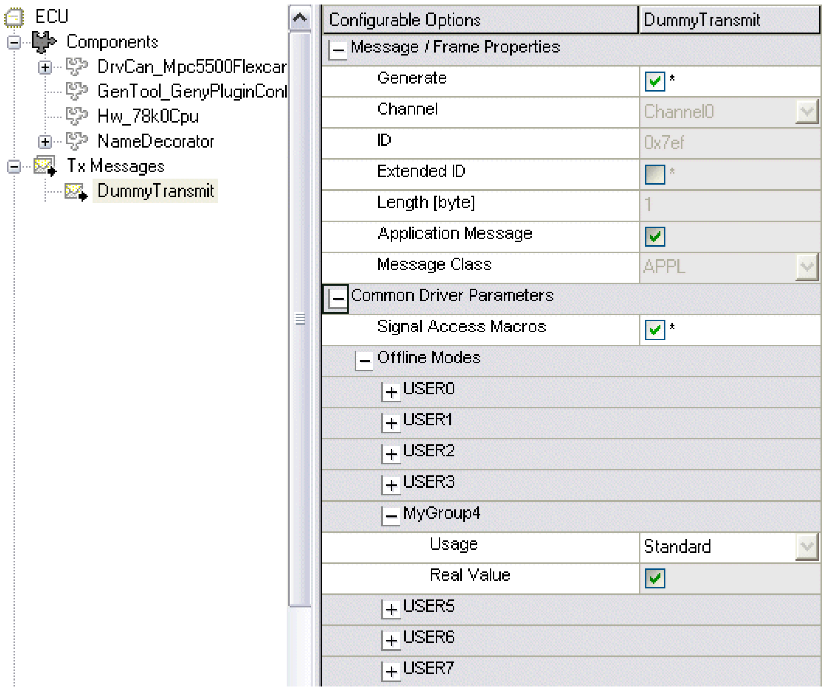
- ⊞ Message Class 2 (NM)
- ⊞ Message Class 3 (TP)
- ⊞ Message Class 4 (DIAG)
- ⊞ Message Class 5 (IL)

Figure 5-5 Partial Offline Mode settings in GENy

Info
You find this information in the configuration view of the CAN Driver.

With the checked checkbox for OfflineMode4 (Message Class 1 (APPL) all application messages are assigned to the Offline Mode 4.

If you select an application message, you will find the following:

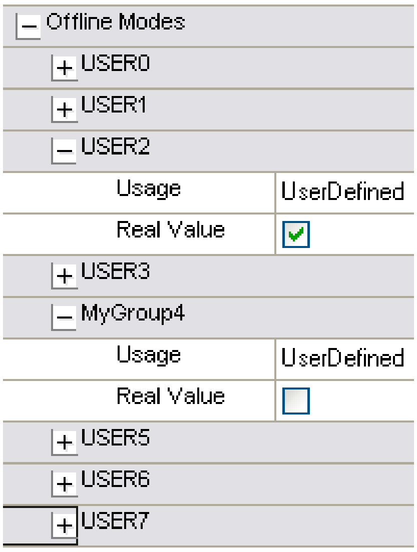


- ECU
  - Components
    - DrvCan_Mpc5500Flexcar
    - GenTool_GenyPluginConf
    - Hw_78k0Cpu
    - NameDecorator
  - Tx Messages
    - DummyTransmit

Configurable Options	DummyTransmit
− Message / Frame Properties	
Generate	☑ *
Channel	Channel0
ID	0x7ef
Extended ID	□ *
Length [byte]	1
Application Message	☑
Message Class	APPL
− Common Driver Parameters	
Signal Access Macros	☑ *
− Offline Modes	
+ USER0	
+ USER1	
+ USER2	
+ USER3	
− MyGroup4	
Usage	Standard
Real Value	☑
+ USER5	
+ USER6	
+ USER7	

Figure 5-6 One Single Application Message Selected

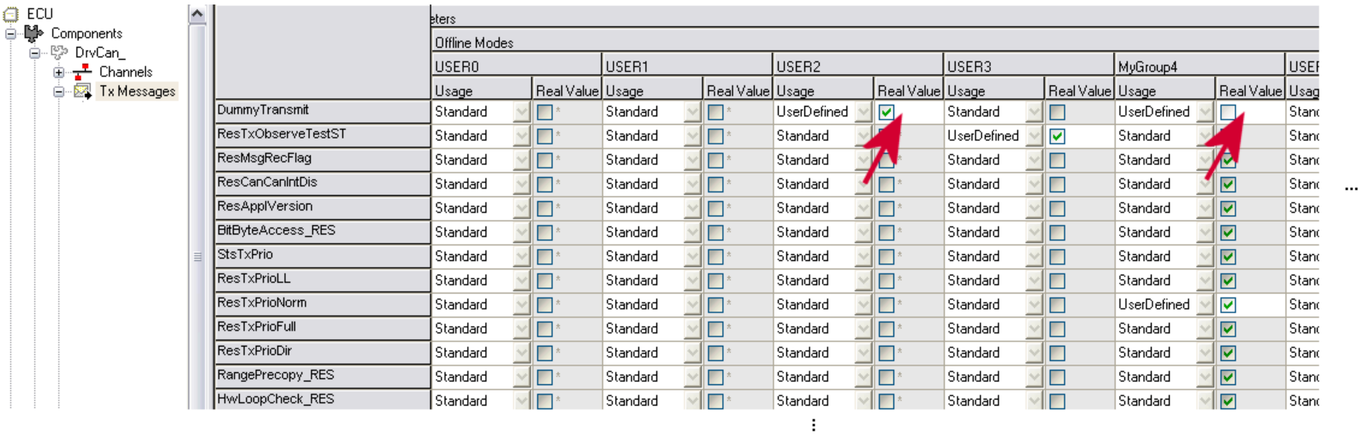
At the Message Class entry you see this is an application message and below you see your MyGroup4 and a checked checkbox. I.e. this message is assigned to MyGroup4.



− Offline Modes	
+ USER0	
+ USER1	
− USER2	
Usage	UserDefined
Real Value	☑
+ USER3	
− MyGroup4	
Usage	UserDefined
Real Value	□
+ USER5	
+ USER6	
+ USER7	

Figure 5-7 User Defined assignment to Offline Modes

For any message you can decide whether to assign the message to another Offline Mode or to additionally assign the message to another Offline Mode. In the example above, the messate DummyTransmit (application message) is not assigned to MyGroup4 anymore. Now this message is assigned to USER2.



ECU
- Components
  - DrvCan_
    - Channels
    - Tx Messages

eters

Offline Modes

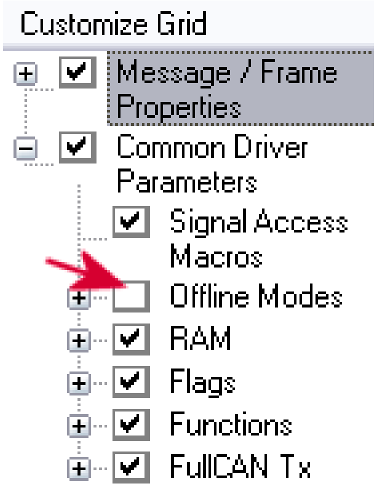
	USER0		USER1		USER2		USER3		MyGroup4		USEF
	Usage	Real Value	Usage	Real Value	Usage	Real Value	Usage	Real Value	Usage	Real Value	Usag
DummyTransmit	Standard	☐ *	Standard	☐ *	UserDefined	☑	Standard	☐	UserDefined	☐	Stanc
ResTxObserveTestST	Standard	☐ *	Standard	☐ *	Standard	☐ *	UserDefined	☑	Standard	☑	Stanc
ResMsgRecFlag	Standard	☐ *	Standard	☐ *	Standard	☐ *	Standard	☐	Standard	☑	Stanc
ResCanCanIntDis	Standard	☐ *	Standard	☐ *	Standard	☐ *	Standard	☐	Standard	☑	Stanc
ResApplVersion	Standard	☐ *	Standard	☐ *	Standard	☐ *	Standard	☐	Standard	☑	Stanc
BitByteAccess_RES	Standard	☐ *	Standard	☐ *	Standard	☐ *	Standard	☐	Standard	☑	Stanc
StsTxPrio	Standard	☐ *	Standard	☐ *	Standard	☐ *	Standard	☐	Standard	☑	Stanc
ResTxPrioLL	Standard	☐ *	Standard	☐ *	Standard	☐ *	Standard	☐	Standard	☑	Stanc
ResTxPrioNorm	Standard	☐ *	Standard	☐ *	Standard	☐ *	Standard	☐	UserDefined	☑	Stanc
ResTxPrioFull	Standard	☐ *	Standard	☐ *	Standard	☐ *	Standard	☐	Standard	☑	Stanc
ResTxPrioDir	Standard	☐ *	Standard	☐ *	Standard	☐ *	Standard	☐	Standard	☑	Stanc
RangePrecopy_RES	Standard	☐ *	Standard	☐ *	Standard	☐ *	Standard	☐	Standard	☑	Stanc
HwLoopCheck_RES	Standard	☐ *	Standard	☐ *	Standard	☐ *	Standard	☐	Standard	☑	Stanc

⋮ ...

Figure 5-8 Overview Messages and Offline Modes

Info

If you cannot find any information concerning Offline Modes you should use the Customize Grid functionality. Activate the view below via: **View|Customize Grid** and then select **Offline Modes**.



Customize Grid

- ☑ Message / Frame Properties
- ☑ Common Driver Parameters
  - ☑ Signal Access Macros
  - ☐ Offline Modes
  - ☑ RAM
  - ☑ Flags
  - ☑ Functions
  - ☑ FullCAN Tx

To get an overall view of which message is assigned to which group, or to do the necessary assignments having a good overview, select all **TxMessages** in the tree view and activate the Offline Modes via Customize Grid (described at the top of this chapter).

5.2.7 Passive State

The CAN Driver's transmit path can be switched to the passive state. In passive state no transmit request is passed to the CAN bus, i.e. no CAN message is sent. However, there is only the CAN bus activity affected but not the Application interface because there is no error code returned and the notification is done in the normal way, i.e. the Application software runs in normal operating mode. This is the main difference to the offline mode. The passive state may be used to localize errors in a CAN bus and is realized by the following CAN Driver service functions

```
void CanSetActive ( void );

void CanSetPassive( void );
```

The passive state of the CAN Driver is usually used during the development phase of the CAN bus. If an Application might disturb the other nodes, it can be switched to passive state temporarily and simulated by an appropriated tool. This is usually done by a Diagnostics.

Info
To use the Passive State efficiently there must be a special support by the network designer. An external tool must be able to take over the tasks of the ECU simultaneously when the ECU is switched to passive state.

The passive state of the CAN Driver must not be mixed up with the passive state of OSEK Network Management. If the OSEK Network Management is put into passive state (service functions SilentNM / TalkNM) only Network Management messages are affected. The passive state of the CAN Driver prevents any CAN messages (including Network Management messages) from being sent on the CAN bus.

Also note the following hints for the usage of the passive state:

- If the passive function is enabled the corresponding code in CanSetPassive() and CanSetActive() is activated, otherwise only dummy macros will be provided. This results in less CAN Driver code and an easy way to switch off this service function without changing the Application software.
- The Application calls the service function CanSetPassive() to prevent transmission. In case of a transmit queue it is cleared, i.e. confirmation activities may be lost during the transition from active to passive state. Beginning with the next CanTransmit() the messages are not sent on the CAN bus until CanSetActive() is called.

  In case of a transmit queue, the service function CanSetPassive() has to be called in the confirmation function of the last message to be sent on the CAN bus. If there is no such request, CanSetPassive() can be called at any time.

  In passive mode, the result seems to be successful, i.e. the code kCanTxOk is returned from CanTransmit(), and all configured flags (cleared by the Application) are set and the functions are called (Common Confirmation Function, Confirmation Flag and/or Confirmation Function). Tx Observation is not used in passive state.
- To restart transmission, the service function CanSetActive() has to be called. Starting with the next call of CanTransmit(), the messages are transmitted again on the CAN bus.

Important
If the CAN Driver is switched from active to passive state, the transmit queue will be cleared and therefore some confirmations may be lost.

5.2.8 Tx Observe

This functionality is used to check the transmit path of the CAN Driver by the following way: After a successful transmit request in the CAN Controller a specific function is called:

```
 void ApplCanTxObjStart( logTxHwObject );
```

If the message was sent on the CAN network successfully another callback function is called in the scope of the transmit interrupt:

```
void ApplCanTxObjConfirmed( logTxHwObject );
```

This functionality can be used to observe any transmission. As the CAN Driver is not time triggered, the call back functions offer the application a way to start a timer with ApplCanTxObjStart and stop this timer with ApplCanTxObjConfirmed. In case of exceeding a predefined time for transmission, the message can be deleted or any other reaction can be done.

In case of a well working system, these callback functions are normally called in a symmetric way within the maximum specified delay time which is allowed in the existing run time environment after a transmit request until the CAN message is sent to the CAN bus successfully. In case of a transmit error a time-out supervision can be implemented by these callback functions and error recovery can be done. If more than one hardware transmit object is used, these callback functions can be called in a nested way and so an additional counter is necessary. That counter has to be reset after each re-initialization of the CAN Controller. This can be done in the following callback function:

```
void ApplCanInit( logTxHwObjectFirstUsed,
logTxHwObjectFirstUnused);
```

5.2.9 Cancellation of a Transmission

There are several ways to cancel a requested transmission.

5.2.9.1 Cancel a Transmission via CanInit

CanInit initializes the CAN controller hardware and can therefore be used to cancel any current transmission. (see Re-Initialization of the CAN Controller). Some controllers do not stop their transmission immediately, so it is possible that the Cancellation via CanInit() could lead to an errorframe on the bus.

5.2.9.2 Cancel a Transmission via CanCancelTransmit or CanCancelMsgTransmit

Both functions work the same way, except that CanCancelTransmit cancels a transmission initiated via CanTransmit and CanCancelMsgTransmit cancels a transmission initiated via CanMsgTransmit.

The call of the confirmation function or the setting of the confirmation flag are suppressed, if this message is already in the transmit buffer of the CAN Controller. If the transmit queue is enabled, a pending transmit request in the queue is canceled.

These functions also delete messages in the hardware transmit buffer if configured. But this feature is strongly dependent of the hardware. Some CAN Driver / CAN Controller require the call of CanRxTask() / CanTxTask() to be able to continue.

Using the cancel functions out of the Tx observe functionality (see above) the handle for the functions must be obtained via the function CanTxGetActHandle(CanObjectHandle logTxHwObject). The return code decides whether it was a CanTransmit or a CanMsgTransmit which causes a CanCancelTransmit or a CanCancelMsgTransmit.

5.2.10 Overview of Transmit Objects

The table shows the naming for different RI versions. Some of the features of the column are hardware dependent.

Names before RI 1.4	Names RI 1.4	Names RI 1.5 and later
Transmit Object	Normal Transmit Object	Normal Transmit Object
Direct Transmit Objects	Full CAN Transmit Object	Full CAN Transmit Object
---	Direct Transmit Objects	---
Dynamic Transmit Objects	Dynamic Transmit Objects	Dynamic Transmit Objects
Low Level Message Transmit	Low Level Message Transmit	Low Level Message Transmit

5.2.11 Normal Transmit Object

A Normal Transmit Object is the hardware transmit object supported by all CAN Drivers. All transmit messages that are not assigned to a Full CAN Transmit Object will be transmitted via this Normal Transmit Object. The transmit queue works only on this object and the Dynamic Transmit Objects can only be transmitted via this object, too.

5.2.12 Full CAN Transmit Objects

Each Full CAN Transmit Object has its own Hardware Transmit Object. This means a Full CAN Transmit Object holds exactly one CAN message with a specific CAN identifier and DLC. These CAN messages are statically assigned by the Generation Tool. Changes of this reference during run time are not possible. There are two reasons for Full CAN Transmit Object:

1. The associated CAN message object is never occupied by another transmit request
2. There is no need to copy the CAN identifier and the DLC. The message data can also be stored directly in the CAN Controller and the transmit request can be initiated directly.

Info
Full CAN objects are sent via CanTransmit() function.

5.2.13 Dynamic Transmit Objects

The CAN Driver supports the transmission of CAN messages with dynamic parameters. These messages must not be specified in the CAN database. This feature can be used in gateways, for example.

These dynamic objects can consist of mixed dynamic and static parts. CAN identifier, DLC and data pointer can be selected separately as dynamic or static. The selection is common for all dynamic objects. Pretransmit functions and confirmation functions are always static.

The CAN identifier priority for dynamic objects is lost if a transmit queue is used. Dynamic objects have a higher internal priority than static objects, independent of their current CAN identifier.

Before the Application can use a dynamic object, the Application needs to reserve one. This can be done by the following service function:

```
CanTransmitHandle CanGetDynTxObj( CanTransmitHandle txHandle);
```

The next step is to set all dynamic parameters of this object. This will be done by calling the service functions:

```
void CanDynTxObjSetId               ( ... );
void CanDynTxObjSetExtId            ( ... );
void CanDynTxObjSetDlc              ( ... );
void CanDynTxObjSetDataPtr          ( ... );
```

After this, the dynamic object can be transmitted by calling CanTransmit(..) with the handle of the dynamic object. The Application is allowed to use a dynamic object several times. If the Application doesn't need the dynamic objects any more, it can be released by the service function

```
vuint8 CanReleaseDynTxObj( CanTransmitHandle txHandle );
```

There are two macros to allow a call of CanReleaseDynTxObj() in a confirmation function. Both macros are only allowed to be called in the context of the user confirmation function of this Dynamic Object.

CanConfirmStart(txHandle)	This macro enables release of dynamic objects in a confirmation function. txHandle has to be equal to the parameter of the confirmation function.
CanConfirmEnd()	This macro restores security mechanism for release of dynamic Objects.

Example:

```
void Confirm_ResDynTxObj ( CanTransmitHandle txHandle )
{
  …
  CanConfirmStart(txHandle);
  if (CanReleaseDynTxObj( txHandle )== kCanDynNotReleased)
  { //error handling }
  CanConfirmEnd();
  …
}
```

If a dynamic object is used several times, the Application has to take care to use the confirmation flag / function.

The maximum number of Dynamic Transmit Objects must be defined statically in the Generation Tool.

Messages of dynamic transmit objects can not be sent via Full CAN Transmit Objects.

5.2.14 Priority of Transmit Objects

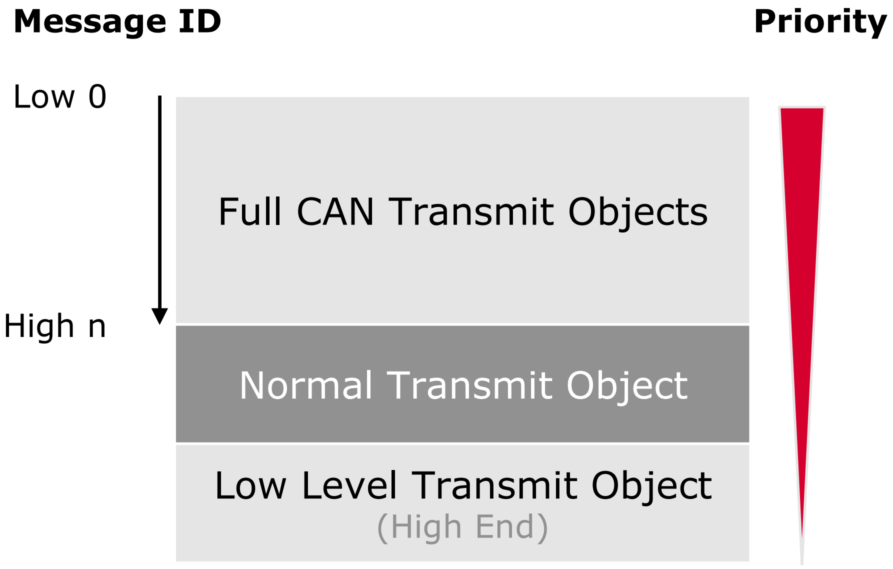


Figure 5-9 Priority of Transmit Objects

Full CAN Objects have the highest priority and they are sorted according to their ID. This is automatically done by the Generation Tool.

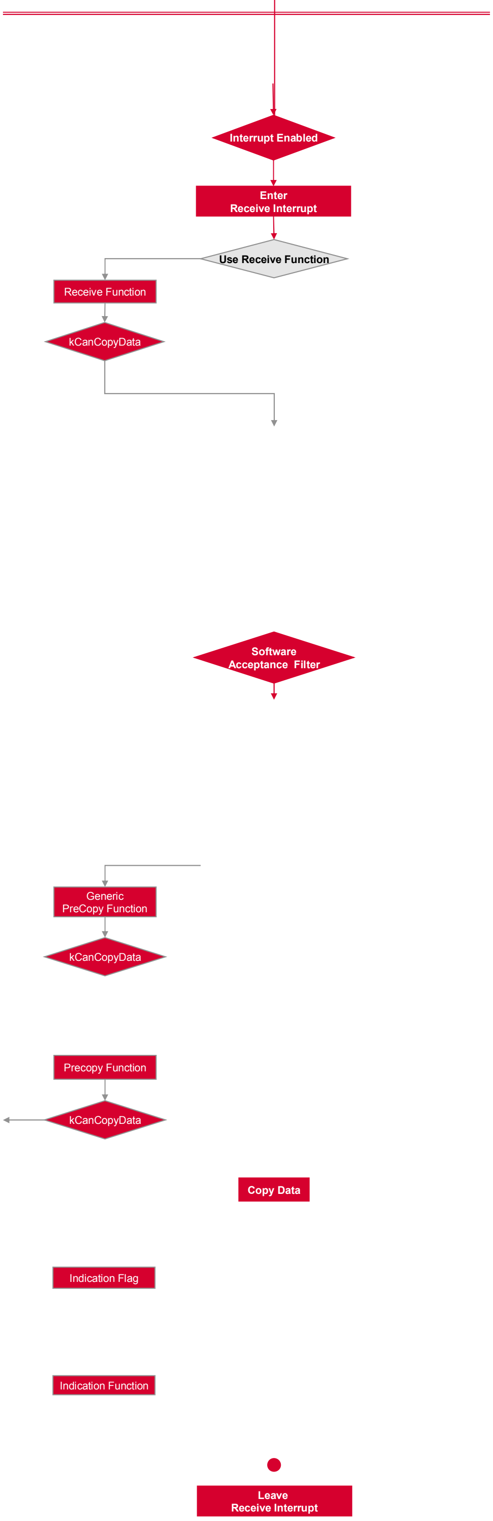
There is only one Normal Transmit Object with a lower priority than the Full CAN Objects. Dynamic Transmit Objects are transmitted via the Normal Transmit Object.

The Low Level Message Transmit Object has the lowest priority.

This priority is only valid, if the hardware is not able to arbitrate according the IDs.

5.3 Reception

5.3.1 Detailed Functional Description



CAN messages are received asynchronously and without any explicit service function call. Normally, the CAN Driver is informed by the CAN Controller via interrupt of the reception of a CAN message. That means the received CAN identifier has passed the hardware acceptance filtering of the CAN Controller and the entire message is stored in a receive register. In case of a Basic CAN object, the message has to be retrieved and processed as fast as possible. If a feature is only in Basic CAN or Full CAN available if is mentioned in the text.
The gray decision symbols branch to features that can be removed from the CAN Driver using the configuration options of the Generation Tool. Disabled features cannot be used for any messages. The code for these features is completely removed. If a feature is enabled, it can be determined for each message whether it is used or not.

The receive callback function ApplCanMsgReceived(..) is called on every reception of a CAN message after the hardware acceptance filter is passed. Within this function the Application may preprocess the received message in any way (ECU specific dynamic filtering mechanisms, gateway functionality, etc...). If the function returns kCanCopyData, the CAN Driver continues the processing. If the function returns kCanNoCopyData, the CAN Driver terminates the message reception.

During the software acceptance filtering (only available for Basic CAN) the CAN Driver first checks for range specific identifiers. For the range specific identifiers special precopy functions may be defined. Afterwards the single CAN identifier based filtering is performed. The CAN Drivers support different mechanisms like linear search, hash search or an index search. In any case the filtering capabilities of the CAN Controller are used. The corresponding receive object has to be determined by comparing the generated CAN identifier in the data description tables with the received CAN identifier in the Basic CAN object.

If the result of the software acceptance filtering is negative (only done for a Basic CAN object), the callback function ApplCanMsgNotMatched() is called. Then the receive interrupt is terminated immediately after the CAN Controller hardware is serviced.

After a CAN identifier match, the DLC will be checked. In case of a failed DLC check there can be a configured callback function to notify the application.

In case of a successful DLC check the generic precopy function is called (if configured). Generic precopy means that a common function named ApplCanGenericPrecopy() is called for all identifiers. If this function returns kCanNoCopyData the CAN Driver terminates further processing. If this function returns kCanCopyData, the CAN Driver continues to work on the message received.

After the generic precopy if configured a precopy function separate for each message according to the entry in the description data is called. Within this user specific function any processing of the message received may occur (complete processing of a message or special storage methods like ring buffers, FIFOs, ...). If the precopy function returns kCanNoCopyData the CAN Driver terminates further processing. If the precopy function returns kCanCopyData, the CAN Driver continues to work on the message received.

In the next step the data is copied to the global data buffer. The CAN Driver copies only the number of bytes from the CAN receive buffer that is stored in the array CanRxDataLen.

Then the indication actions defined for this message are performed. This means the indication flag is set and/or the indication function is called. The Application has to reset the indication flag before or after data processing.

In the following sections the processing steps are described using sequence charts. The vertical directed lines within these diagrams represent program objects like interrupt routines, functions or data objects. The horizontal lines represent program flow or data access within the program. Within the sequence charts below flow control and program instances are described using thick lines, data access is described using thin lines. Time flows from the top of a chart downwards so that sequence „1" is performed before sequence „2". The description of the sequence charts is given in the tables following the charts.

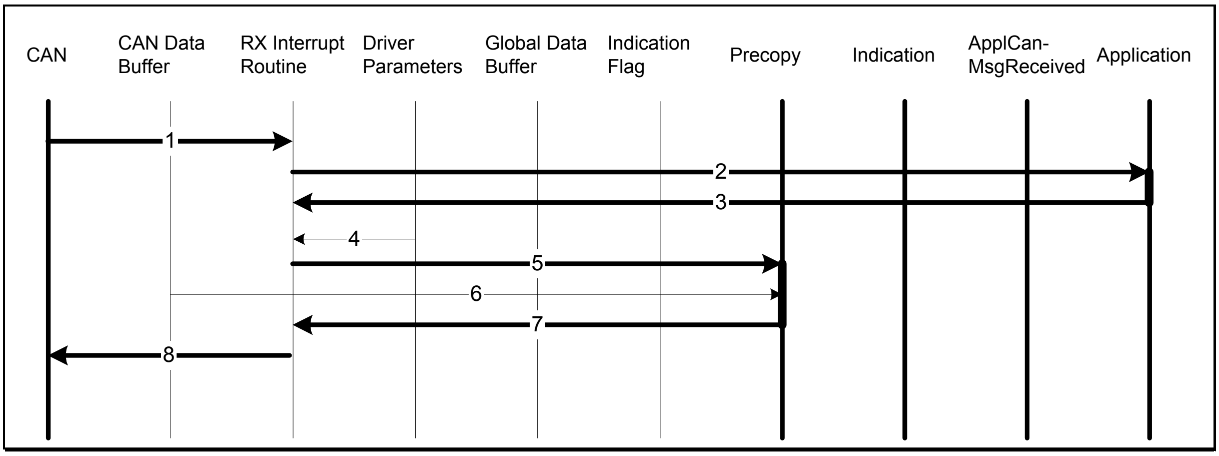


Figure 5-11 Reception of a CAN message: The data is completely processed in the precopy function

No	Description
1	A CAN message has passed the hardware acceptance filtering, the receive interrupt routine is triggered
2	If configured, the ApplCanMsgReceived() callback function is called
3	The ApplCanMsgReceived() callback function returns kCanCopyData
4	Software acceptance filtering and identification of the received CAN message
5	If configured, the precopy function is called. The Application is able to take control over the receive process immediately after the software acceptance filtering and direct access to the CAN Controller receive register is possible.
6	Within the precopy function the data in the CAN Controller hardware registers are read and completely processed.
7	The precopy function returns kCanNoCopyData. No further processing (copying of data, indication actions) occurs in the CAN Driver
8	After servicing the CAN Controller hardware (the receive registers of the CAN Controller are released), the receive interrupt routine is left.

Info

1. If the ApplCanMsgReceived() callback function returns kCanNoCopyData, the received message is ignored. This means no further software filtering, no precopy, no

	copying of data and no indication actions are performed. 2. If the precopy function returns kCanNoCopyData, no copying of data and no indication actions are performed.

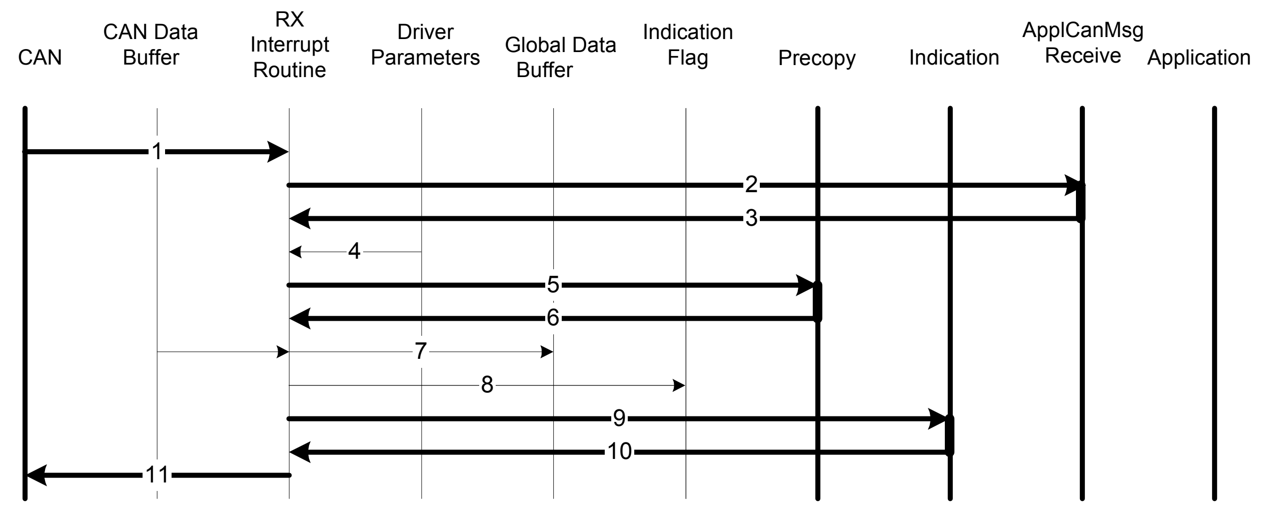


Figure 5-12 Reception of a CAN message: The CAN Driver internal copying mechanism is used

No	Description
1	A CAN message has passed the hardware acceptance filtering, the receive interrupt routine is triggered
2	If configured, the ApplCanMsgReceived(..) callback function is called
3	The ApplCanMsgReceived(..) callback function returns kCanCopyData
4	Software acceptance filtering and identification of the received CAN message
5	If configured, the precopy function is called. The Application is able to take control over the receive process immediately after the software acceptance filtering and the direct access to the CAN Controller receive register is possible.
6	The precopy function returns kCanCopyData. The CAN Driver continues its normal processing.
7	The received data are copied from the CAN Controller receive register to the global data buffer associated to the CAN message
8	If configured, the indication flag is set (must be reset by the Application)
9	If configured, the indication function is called; any user actions can be performed within this user specific function
10	Indication function returns to the receive interrupt routine
11	Receive interrupt routine is left

5.3.2 Receive Function

Before the software filtering is done, the Application optionally may use the ApplCanMsgReceived() callback function called by the CAN Driver. Within this function the Application can define whether to process the message received or not.

5.3.3 Range-Specific Precopy Functions

The CAN Driver's receive path can be configured to filter special identifier ranges and associated precopy functions will be called directly. Up to four ranges are supported by the CAN Driver. The ranges must be defined by a start address (e.g. 0x400) and a mask (e.g. 0x1F, i.e. if a bit is set it means don’t care) and leads to a specific range (in our example it is from 0x400 to 0x 41F). The ranges are typically predefined by the Generation Tool for special functions. If these are not used they are available for the application:

Range 0	Network Management	If the usage of a Network Management is configured
	Application	Application specific. May be used by the Application
Range 1	Diagnostics	If extended addressing mode of the Transport Protocol is configured
	Application	Application specific. May be used by the Application
Range 2	Special usage	Car manufacturer specific
	Application	Application specific. May be used by the Application
Range 3	Application	Application specific. May be used by the Application

Special capabilities of some CAN Controllers with several hardware acceptance filters may also be used for the range specific filtering.

5.3.4 Identifier Search Algorithms

The following software filtering mechanisms are supported: All mechanisms but linear are optional in the different hardware implementations.

Linear Search: The identifier of the incoming message is compared to all CAN identifiers in a table (if found, the search stops). The average search time is proportional to the number of receive messages.

Hash Search: An optimized search algorithm with a small known number of search steps. The Generation tool calculates an optimized search table and some parameters used at run time. The number of search steps can be defined by the user. The less search steps the bigger the resulting hash tables.

Table Search: This is a kind of hash mechanism. The last three bits of a CAN identifier are used as a selector for the search table. There are 8 different tables for each of the hardware acceptance filters in the CAN Controller. Within the table a linear search is implemented.

Index Search: A table with 2048 entries (one entry for each identifier) is used for software filtering. Index Search is used for Standard ID only.

5.3.5 DLC check

The Data Length Code of a received message will be compared to the length of the Application receive buffer of this message. If the DLC is smaller than the Application receive buffer, data will not be copied. The length of the received message buffer is the maximum length which is necessary to treat all signals for this ECU. To inform the application the callback function ApplCanMsgDlcFailed will be called. The reception process will be terminated afterwards.

Depending on the OEM the length of the received data bytes can be different at run time. It is also possible to compare the length of the received message with a minimum length which can be smaller than the Application receive buffer.

The behavior can be configured via generation tool and the database attribute GenMsgMinAcceptLength.

5.3.6 Data Copy Mechanism

There are two different methods for the Application to access the data received from the CAN bus.

5.3.6.1 Internal

Using the internal data copy mechanism, the CAN Driver copies the contents of the CAN controller receive registers to a global data buffer associated to the receive message. The Application can access the signal values in the message specific data buffer using access macros or functions. The access macros are generated by the Generation Tool using information in the CAN database. The signal access macros always return **unsigned values**.

The Application itself is responsible for the data consistency of signals in a CAN message which cannot be handled in atomic operations because the receive buffer may be overwritten asynchronously by a CAN receive interrupt. Different mechanisms can be used to guarantee data consistency:

1. Disabling of the CAN receive interrupt.
2. Read the receive signal. Compare the signal value with the signal in the hardware buffer. Repeat the read operation if the values differ.
3. Usage of the message based indication flag:
   3.1 Clear the message indication flag
   3.2 Read the data (one or more signals of a message)
   3.3 Check the message indication flag: If set then return to 3.1

Depending on the OEM the length of the received data bytes can be different at run time. Instead of copying all needed bytes (equal to the length of the global data buffer associated to the receive message) the CAN Driver can be configured to copy the number of received bytes. In case the number of received bytes exceed the length of the data buffer, the CAN Driver takes care to copy at maximum the length of the data buffer.

Info
The signal access macros are not affected. The application has to make sure, that it does not access data via access macros that is not copied now because of a change of the data length.

5.3.6.2 User-defined Precopy Functions

The user can define specific precopy functions for each receive object in the Generation Tool. If defined, the CAN Driver calls this user-specific function immediately after the software filtering. Within this precopy function the Application can access the data directly in the CAN Controller receive registers. The precopy function indicates this to the CAN Driver by the appropriate return code kCanNoCopyData and the further processing will be terminated immediately. On the other side the CAN Driver can be forced to continue with normal processing of the message after the precopy function by using the return code kCanCopyData.

The parameter of the precopy function is a pointer to a structure. This structure includes the handle of the received message and a pointer to the received data.

A separate user-specific function may be defined for each receive message. But it is also possible to use the same function for different messages.

If no such function is defined, a NULL pointer is written to the corresponding description data by the Generation Tool.

The user has to note that these user-specific functions are called in the receive interrupt. Only short receive actions should be done to avoid negative influence on the Application task by a long interrupt disable time.

The precopy mechanism can be used to handle only a small number of receive signals in an efficient way, if there is a CAN receive message with 8 bytes but the receiving ECU for example only needs the 6th bit of the 7th byte. The standard copy routine starts always at the beginning of the receive data buffer and copies all data up to the last byte with significant signals for the dedicated node (the 7th byte in the example above). This results in some overhead in RAM and run time, particularly if these signals are mapped in the rear part of the message. The precopy function can therefore be used to implement a user specific copy routine and has to return the return code kCanNoCopyData.

Another example for a precopy function is a compare mechanism between the CAN Controller receive register and the global Application buffer. If both are matching, data have not to be copied and the indication is not necessary, i.e. kCanNoCopyData is returned. Otherwise the return code kCanCopyData leads to the standard copy mechanism of the CAN Driver and notification to the Application using indications.

The precopy function can also be used to implement receive queues (FIFO, FILO or ring buffer).

5.3.7 Notification

After the reception of a message from the CAN bus and the successful hardware and software acceptance filtering, the Application can be notified by different indication mechanisms:

5.3.7.1 Data Interface (Indication Flag)

If an indication flag is used, this message related flag is set by the CAN Driver, if the associated CAN message was received. This is done in the scope of the receive interrupt. The flag must be cleared by the Application.

⚠	**Caution** Interrupts have to be disabled during reset of the indication flags because of the read-modify-write conflict if this operation is interrupted by a CAN receive interrupt routine. This can result in the loss of events.

5.3.7.2 Functional Interface (Indication Function)

In parallel or instead of the data interface a functional interface can be configured, i.e. a user-specific function is called if the associated CAN message was received. This is also done in the scope of the receive interrupt and therefore special care on the run time of this user-specific function has to be taken. A special notification mechanism for the Application can be implemented in such an indication function.

5.3.8 Not-Matched Function

If a CAN message has passed the hardware acceptance filtering but is rejected by the software acceptance filtering (in case of a Basic CAN receive object) a special callback function will be called (if configured):

```
void ApplCanMsgNotMatched( ... );
```

5.3.9 Overrun Handling

An Overrun appears if a CAN message is lost in the Basic CAN receive object, because the other was not treated yet entirely. There are two possibilities how a message could be lost. In some cases the old message was overwritten with a new message. In other cases a new message couldn’t be received.

If enabled, the Application has to provide an overrun callback function:

```
void ApplCanOverrun( void );
```

The overrun handling itself is done by the CAN Driver.

5.3.10 Full CAN Overrun Handling

A Full CAN Overrun appears if a CAN message is lost in the Full CAN receive object, because the other was not treated yet entirely. There are two possibilities how a message could be lost. In some cases the old message was overwritten with a new message. In other cases a new message couldn’t be received.

If enabled, the Application has to provide an overrun callback function for Full CAN objects:

```
void ApplCanFullCanOverrun( void );
```

The overrun handling itself is done by the CAN Driver

5.3.11 Conditional Message Received

The Conditional Message Received function ApplCanMsgCondReceived() will be conditional called for each reception of a CAN message. The condition can be set / reset and read by application via CanResetMsgReceivedCondition(), CanSetMsgReceivedCondition(), and CanGetMsgReceivedCondition(). The condition is automatically set by CanInitPowerOn() and CanSleep().

5.4 Bus-Off Handling

There are several functions provided by the CAN Driver to handle a BusOff state of the CAN Controller after severe transmit errors. For some CAN Controllers a re-initialization must be done to satisfy the hardware requirements others are changing automatically to the 'Error Active' state after 128 x 11 recessive bits on the CAN bus as it is specified in the CAN protocol. Nevertheless it is recommended by most of the customer specific CAN bus specifications to re-initialize the CAN Controller in every case, because the transmit error might be caused by a faulty bit in the CAN Controller registers, e.g. bus timing registers, in case of EMC influences. The following service functions have to be used by the Application to handle a BusOff error:

```
void CanResetBusOffStart( CanInitHandle initObject );

void CanResetBusOffEnd( CanInitHandle initObject );
```

Typically an extension (compared to the CAN protocol specific requirements) of the error recovery time for the CAN bus is implemented. This is done by switching the CAN Driver's transmit path to off using the service function CanOffline(). Because of recursive calls of some CAN Driver service functions, CanResetBusOffStart(..) and CanResetBusOffEnd(..) are only allowed to be called in the offline mode of the CAN Driver, i.e. CanOffline() has to be called before.

Typically the Network Management handles BusOff errors. In such case there are no additional activities necessary by the Application. If no Network Management is used, the Application has to provide a callback function

```
void ApplCanBusOff( void );
```

This callback function is called by the CAN Driver in case of BusOff. The error handling as described above has to be done in this function. CanOnline has to be called outside of this function on task level.

Important
For CAN controller which has autorecovery after BusOff detection we don't recommend to use the status polling. If using status polling with autorecovery it could happen, that the application doesn't detect a BusOff because a transmit request was detected first and the application wasn't informed about the BusOff.

5.5 Sleep Mode

Some CAN Controllers support a special power-down mode with reduced power consumption which is typically called sleep mode. This mode will be entered by the following service:

```
vuint8 CanSleep( void );
```

Important

Before entering the sleep mode, some hardware specific preconditions have to be ensured, e.g. the CAN Controller transmit registers have to be empty. It has to be guaranteed, that the following service functions are called before CanSleep():

void CanOffline(void);

void CanResetBusSleep(CanInitHandle initObject);.

The return to normal mode will be initiated by an explicit request of the Application:

```
vuint8 CanWakeUp( void );
```

Sleep mode is not supported by all CAN Controllers. If not, both related service functions are provided to guarantee a unified service function interface for all CAN Drivers and to make the Application mostly hardware independent. However, the functions itself have no effect on the CAN Controller.

A subset of CAN Controllers, which are supporting a sleep mode in principle, are able to be awakened by any CAN bus activity, i.e. a dominant level on the CAN bus. This wake-up by CAN is an asynchronous event, normally detected by a special wake-up interrupt. The Application will be notified by the following callback function:

```
void ApplCanWakeUp( void );
```

This callback function has to be provided by the Application. CanWakeUp() doesn't have to be called in this case, because the CAN Controller returns to normal mode automatically or initiated by the CAN Driver before this function call. Other communication related issues like the activation of the bus transceiver hardware used or the return to the online mode (see CanOnline()) have to be done in this callback function or as a consequence of this event.

If a CAN Controller doesn't support a wake-up by the CAN bus, other hardware substitutions like an external interrupt based on the CAN Controller's Rx line have to be implemented.

The application should check the return value of `CanSleep()` and `CanWakeUp()` in every case to get the status of the CAN Controller. If `CanSleep()` returns `kCanFailed` the CAN controller hasn’t entered into sleep mode. If `CanWakeUp()` returns `kCanFailed` the CAN controller has not woken up. The application has to decide how to react on this behavior.

If sleep mode is not entered, no CAN wake-up interrupt will be generated on detection of any message on the CAN bus. The callback function ApplCanWakeUp() will not be called and as a consequence the bus transceiver will not be initialized. This may lead to a deadlock. Therefore it is necessary to call CanSleep() successfully to build a wake-up capable system.

There is a limitation in the access to the API in Sleep mode.

The implementation of this functionality is very hardware dependent. See also CAN controller specific documentation TechnicalReference_CAN_<hardware>.pdf [#hw_sleep].

5.6 Special Features

5.6.1 Status

Some internal software states of the CAN Driver and hardware states of the CAN Controller can be read by the return code of the following service function:

```
vuint8 CanGetStatus( void );
```

In detail this is the following information:

- CAN Controller is in sleep mode (CanSleep() was called)
- CAN Controller is in stop mode (CanStop() was called)
- CAN Driver transmit path is in offline mode(CanOffline() was called)
- current error states of the CAN Controller (Error-Active, Warning, Error-Passive or Bus-Off)

Not all of the CAN protocol specific bus states are supported by each CAN Driver. Please refer to the CAN Controller related section of the CAN Driver documentation for details TechnicalReference_CAN_<hardware>.pdf [#hw_status].

There are special macros to provide an easier access on the single information in the return code. These macros are true (not equal to 0) if the specific condition is valid and false (equal to 0) if not. The parameter of this macros is the status, i.e. the return code of CanGetStatus():

```
vuint8 CanHwIsOk      ( vuint8 status );
vuint8 CanHwIsWarning( vuint8 status );
vuint8 CanHwIsPassive( vuint8 status );
vuint8 CanHwIsBusOff ( vuint8 status );
vuint8 CanHwIsSleep  ( vuint8 status );
vuint8 CanHwIsWakeUp ( vuint8 status );
vuint8 CanHwIsStop   ( vuint8 status );
vuint8 CanHwIsStart  ( vuint8 status );
vuint8 CanHwIsOffline( vuint8 status );
vuint8 CanHwIsOnline ( vuint8 status );
```

If the hardware status information isn’t used by the Application this part of the functionality can be disabled.

5.6.2 Stop Mode

The function CanStop() switches the CAN controller hardware to a state in which the CAN controller doesn't influence the communication of other nodes on the bus. For example no hardware acknowledge is given, messages can't be transmitted or received. In this state the Can controller can't be activated by activities on the CAN bus.

The function CanStart() reactivates the CAN controller hardware again.

The implementation of this functionality is very hardware dependent. See also CAN controller specific documentation TechnicalReference_CAN_<hardware>.pdf [#hw_stop].

5.6.3 Remote Frames

The CAN Driver ignores remote frames and doesn't answer on a remote request.

5.6.4 Interrupt Control

The interrupt control of the CAN Driver is done by the service functions

```
void CanGlobalInterruptDisable( void );

void CanGlobalInterruptRestore( void );
```

These functions have been moved to VstdLib. Only macros for compatibility reasons are still provided in the CAN Driver:

```
#define  CanGlobalInterruptDisable         VStdSuspendAllInterrupts
#define  CanGlobalInterruptRestore          VStdResumeAllInterrupts
```

...more information see in the technical reference of the VStdLib. (TechnicalReference_VstdLib.pdf).

5.6.4.1 Security Level

The security levels can be used to guarantee the data consistency of a complete CAN message during the copy process (this is a must, because the CAN Driver does not know anything about the signal structure of the message) and the access to the notification flags (indication and confirmation). During these operations the interrupt lock time is as short as possible. Depending on the program scope with access to CAN message signals, indication or confirmation flags in the Application the following actions in the CAN Driver have to be realized without any interruption:

- Copy process for receive messages (in the scope of the receive interrupt)
- Copy process for transmit messages (in the scope of CanTransmit(..) or in a pretransmit function)
- Set of indication and confirmation flags (in the scope of the receive and transmit interrupt)
- Some internal mechanisms for data consistency.

Therefore different security levels are supported:

Level	CAN Driver prevention	Restrictions for the Application
00	None	No consistency mechanisms at all. The CAN driver has to be configured to polling mode. CAN interrupts are not allowed. All CAN driver tasks, all calls to service functions, all data and flag access must be performed from the same level.
10	No Flag and Copy Security	No usage of CAN transmit and receive signals in the interrupt context. Usage of the TxQueue is allowed in Tx polling mode only. No reset of notification flags in the interrupt context If a fully-preemptive operating system is used, the access to the transmit data and transmission of the data has to be done on the same priority level. (data consistency).
20	Interrupts are disabled during the copy process of transmit messages	a) Interrupt-Mode: No usage of CAN receive signals and no reset of notification flags in the interrupt context b) Polling-Mode Access to CAN receive signals, indication and confirmation flags is only allowed at the same level or at lower level than CanTask().
30 (default)	Interrupts are disabled during the copy process of transmit and receive messages and during the access to the notification flags	No restrictions for the Application, neither on the usage of CAN receive or transmit signals nor on the reset of notifications flags, can i.e. both be done at any time.

Figure 1: Security levels

Important
Be careful if a pretransmit function is used. Interrupts are not disabled during the call of this user specific function by the CAN Driver, therefore the restrictions for security level 10 are valid. If the interrupts are not disabled before and restored after the copy process by the Application, data consistency of a CAN messages cannot be guaranteed if the transmit queue is used..

5.6.4.2 Control of CAN interrupts

The interrupt control of the CAN Interrupts is done by the service functions

```
void CanCanInterruptDisable( void );
void CanCanInterruptRestore( void );
```

These service functions control the CAN Interrupts. CanCanInterruptDisable disables the CAN interrupts and CanCanInterruptRestore restores the state of the CAN interrupts before the call of CanCanInterruptDisable. This mechanism is accompanied with a counter to recognize the number of calls. A “disable” increments the counter and a “restore” decrements the counter to allow nested calls of these functions.

These functions could only be called as pair. That means that on a CanCanInterruptDisable must follow a CanCanInterruptRestore. Otherwise the selected interrupt(s) are always disabled.

Additionally refer to the hardware description for the specific platform TechnicalReference_CAN_<hardware>.pdf [#hw_int] especially concerning the handling of the wake-up interrupt. It depends on the hardware whether the wake-up interrupt is included or not.

There are two call back functions for the application. After the CanCanInterruptDisable the function ApplCanAddCanInterruptDisable is called and after the CanCanInterruptRestore the function ApplCanAddCanInterruptRestore is called.

Use these two functions to handle the wake-up interrupt if the hardware treats this interrupt separately or if the Driver runs in Polling Mode disable the polling tasks.

To activate the call back functions refer to the API description of the functions.

5.6.5 Assertions

To detect some incorrect internal conditions of the CAN Driver during development, integration and software test, there are different categories of so called assertions configurable:

- User interface (for example input parameters, reentrance if not allowed)
- Fatal hardware errors
- Generated data
- Internal software errors (for example inconsistent internal states)

Each type of assertion can be configured independently.

These assertions will help in different development phases to deal with unexpected problems which cannot be handled by the CAN Driver internally. In such case the following callback function will be called by the CAN Driver:

```
void ApplCanFatalError( vuint8 errorNumber );
```

This callback function has to be provided by the Application. The function parameter errorNumber gives more detailed information about the kind of error which is occurred.

Generally, the error number has to be checked to solve the underlying problem.

⚠	**Important** This callback function must not return to the CAN Driver afterwards.

The recommended usage of the different assertion categories is explained in the following table:

User Interface	Development of Application software
Fatal hardware errors	Development of Application software New CAN Controller used
Generated Data	New version of the Generation Tool used Test of software changes in the Generation Tool or CAN Driver (Vector internal)
Internal software errors	Test of software changes in the CAN Driver (Vector internal)

These checks could be very run-time intensive and should only be activated for the development phase of the CAN Driver

The call back function *ApplCanFatalError()* is called with the following error codes:

In case of a user assertion:	
kErrorTxDlcTooLarge	*CanTransmitVarDlc()* or *CanDynTxObjSetDlc()* called with DLC > 8.
kErrorTxHdlTooLarge	service function called with transmit handle too large
kErrorIntRestoreTooOften	*CanCanInterruptRestore()* called too often
kErrorIntDisableTooOften	*CanCanInterruptDisable()* called too often
kErrorChannelHdlTooLarge	service function called with channel handle too large
kErrorInitObjectHdlTooLarge	*CanInit()* called with parameter “initObject” too large
kErrorTxHwHdlTooLarge	*CanTxGetActHandle()* called with logical hardware handle too large
kErrorHwObjNotInPolling	*CanTxObjTask(), CanRxFullCANObjTask()* or *CanRxBasicCANObjTask()* called for hardware object which is configured to interrupt mode.
kErrorHwHdlTooSmall	*CanTxObjTask(), CanRxFullCANObjTask()* or *CanRxBasicCANObjTask()* called for hardware object handle too small
kErrorHwHdlTooLarge	*CanTxObjTask(), CanRxFullCANObjTask()* or *CanRxBasicCANObjTask()* called for hardware object handle too large
kErrorAccessedInvalidDynObj	*CanGetDynTxObj(),CanReleaseDynTxObj()* or *CanDynTxObjSet...()* is called with wrong transmit handle (transmit handle too large)
kErrorAccessedStatObjAsDyn	*CanGetDynTxObj(),CanReleaseDynTxObj()* or *CanDynTxObjSet...()* is called with wrong transmit handle (transmit handle belongs to a static object)
kErrorDynObjReleased	*UserConfirmation()* or *UserPreTransmit()* is called for a dynamic object which is already released.
kErrorPollingTaskRecursion	CAN Driver Polling tasks (*Can...Task()*) are called

	recursive or interrupt each other.
kErrorDisabledChannel	Service function called for disabled channel on systems with multiple configurations.
kErrorDisabledTxMessage	*CanCancelTransmit() or CanTransmit()* called with txHandle that is not active in the current configuration. (Physical multiple ECU)
kErrorDisabledCanInt	*CanSleep()* or *CanWakeUp()* is called with disabled CAN Interrupts (via *CanCanInterruptDisable()*).
kErrorCanSleep	*CanStop(), CanCanInterruptDisable()* or *CanCanInterruptRestore()* called during Sleep mode, or offline mode is not active during sleep mode.
kErrorCanOnline	*CanSleep()* or *CanStop()* is called without offline mode.
kErrorCanStop	*CanSleep()* is called during Stop mode or offline mode is not active during Stop mode.
kErrorWrongMask	*CanSetTxIdExtHi()* is called with illegal mask (mask higher than 0x1F).
kErrorWrongId	*CanDynTxObjSetId() or CanDynTxObjSetExtid()* is called with illegal ID (standard ID higher than 0x7ff or extended ID higher than 0x1FFFFFFF).
In case of a generation assertion:	
kErrorTxROMDLCTooLarge	Error in generated table of transmit DLCs
In case of a hardware assertion:	
kErrorTxBufferBusy	HW transmit object is busy, but this is not expected
In case of a internal assertion:	
kErrorTxHandleWrong	saved transmit handle has an unexpected value
kErrorInternalTxHdlTooLarge	internal function called with parameter tx handle too large
kErrorRxHandleWrong	The variable rx handle has an illegal value.
kErrorTxObjHandleWrong	The handle of the hardware transmit object has an illegal value.
kErrorReleasedUnusedDynObj	*CanReleaseDynTxObj()* is called for an object which is already released.
kErrorTxQueueToManyHandle	The data type of the Tx Queue cannot handle all tx messages.
kErrorInternalChannelHdlTooLarge	Static function called with channel handle too large or calculated channel handle too large.
kErrorInternalDisabledChannel	Static function called for disabled channel on systems with multiple configurations.
kErrorInternalDisabledTxMessage	Confirmation called with txHandle that is not active in the current configuration. (Physical multiple ECU)

See the CAN Controller specific part of the CAN Driver documentation TechnicalReference_CAN_<hardware>.pdf [#hw_assert] to get the list of additional hardware specific error numbers for each CAN Driver.

5.6.6 Hardware Loop Check

There are two kinds of handling loops in the CAN Driver internally. The first one uses a counter or other mathematics algorithms to abort the loop. The second one uses hardware information from the CAN Controller to abort the loop.

Some of these state transitions have to be done by two steps:

1. Request
2. Acknowledge

In the first step the request for a specific action (e.g. re-initialization of the CAN Controller) is set but generally it cannot be entered immediately because of the prerequisite that the CAN bus has to be in idle state, i.e. waiting for a recessive CAN bus level. In normal operation the described behavior is non-critical. However, an exception is a malfunction of the hardware. If the hardware is damaged or disturbed for a longer time, this loop may be too long or even endless and is finally stopped by a watchdog reset. Because of this restrictive error recovery the complete software functionality is affected, nothing can be done to prevent the repetition and additionally it is not possible to store any error specific diagnostic information, i.e. the problem cannot be checked later.

To avoid those kinds of endless loops, the user can configure a special loop control. This has to be handled by the Application. It cannot be done by the CAN Driver itself because it is hardware dependent.

Therefore the Application is informed once by the following callback function if such a critical loop is entered:

```
void ApplCanTimerStart( vuint8 timerIdentification );
```

This callback function starts a timer realized by the Application. The recommended timer handling is counting downwards to zero because of faster code on most microprocessors. The parameter identifies the timer, i.e. the kind of loop. It is necessary to identify the loop type because the corresponding start value has to be set. Beside of this, different (not the same) loops can be started re-entrant and so the Application has to provide one timer for each kind of loop. The list of necessary timers is pre-defined by the CAN Driver and depends on the CAN Controller. Please refer to CAN Controller specific documentation for a detailed list TechnicalReference_CAN_<hardware>.pdf [#hw_loop].

During the loop wait state a second callback function is called repeatedly to control the break condition for the loop by the Application:

```
vuint8 ApplCanTimerLoop( vuint8 timerIdentification );
```

This callback function returns the status of the corresponding timer to the CAN Driver. The return code must be TRUE (not equal to 0) if the timer is still running and FALSE (equal to 0) if the timer has expired. In this case the CAN Driver loop will be left immediately. The Application must be aware of a serious problem in the hardware and the following actions have to be done:

- Store diagnostics information
- Switch off transmit (CanOffline()) and receive path of the CAN Driver

- Re-initialization of the CAN Driver (CanInit()). This may lead to the next loop control failure, therefore it has to be limited and in case of a permanent severe hardware problem a special limp home state has to be foreseen.

If the loop is terminated, a third callback function is provided to stop the previously started loop control timer:

```
void ApplCanTimerEnd( vuint8 timerIdentification );
```

⚠	**Important** Be aware of the priorities of the timer interrupt routine and the CAN interrupt routine. If the priority of the timer interrupt is below the CAN Interrupt priority the timer value for the loop check may not be changed anymore while a CAN interrupt routine is running.

5.6.7 Support of OSEK-Compliant Operating Systems

If an OSEK operating system is used (ISR category 2), the hard-coded interrupt routines for receiving, transmitting, error and wake-up are replaced by the ISR macro. In this case an OSEK-specific header has to be included in can_inc.h to provide this macro.

5.6.8 Multiple-Channel CAN Driver

There are two different kinds of multiple-channel CAN Drivers: Sometimes two CAN Controllers are used by one ECU on the same CAN bus, to increase the number of receive and transmit objects. Logically, they can be conceived as a single CAN Controller. This behavior is described in the chapter Common CAN. more...

Usually, two (or more) CAN Controllers are used to serve different CAN networks, for example in gateways.

5.6.8.1 Indexed CAN Driver

Indexed CAN Drivers work on more than one CAN bus without doubling of code. Function names are equal to single channel (standard) CAN Driver. Function parameter are different in many cases.
Switches in can_cfg.h are without a suffix but with effect to all CAN channels.

5.6.9 Standard Polling Mode

In polling mode no interrupts are used. Instead of interrupts the Application has to call cyclic service functions in the CAN Driver, to work on transmit and receive messages as well as other asynchronous events. This cyclic service function is

```
void CanTask ( void );
```

and calls all needed service functions for transmission, reception, error and wake-up which can also be polled separately by the following service functions:

```
void CanRxBasicCANTask  ( void );
void CanRxFullCANTask   ( void );
void CanTxTask        ( void );
void CanErrorTask  ( void );
void CanWakeUpTask ( void );
```

The transmission and the reception of CAN messages can be served by interrupt or by polling separately. Several configurations for polling are available:

- Full CAN Receive objects (for Full CAN Controllers only)
- Basic CAN Receive Objects
- Transmit objects
- Errors
- Wake-Up

5.6.9.1 Application Hints

Concerning the transmit polling the handling depends on the configuration of transmit queue and the confirmation notification:

- No transmit queue but confirmation flags and/or confirmation functions are configured: The CanTxTask() has to be called cyclically as fast as the confirmation notification is needed or before CanTransmit() is called to release the CAN Controller hardware transmit register.
- Transmit queue is configured: CanTransmit() puts only a transmit request into the transmit queue. CanTxTask() transmits the messages on the CAN bus and does the confirmation as well. Therefore CanTxTask() has to be called as fast as confirmation is needed and the messages should be transmitted.

5.6.10 Handling of different identifier types

Every Vector CAN Driver supports per default only the standard mode using 11 bits for a CAN identifier. In addition to this standard mode, some Vector CAN Drivers also support the feature of extended mode using 29 bits for a CAN identifier.

Depending on the selected mode (standard or extended CAN identifiers) the Generation Tool switches to the correct initialization structures used for the corresponding mode. The type and number of supported search algorithms depends on the mode. Four different CAN Driver configurations are possible:

- Standard mode (only 11 bit CAN identifier)
- Extended mode for the normal receive path of single CAN messages (only 29 bit CAN identifier)
- Mixed mode (11 bit and 29 bit CAN identifier mixed on one CAN bus)
- For indexed drivers a bus dependent mode (11 bit CAN identifier on one and 29 bit CAN identifier on the other CAN bus).

5.6.11 Copying Mechanisms

CanCopyToCan or CanCopyFromCan are hardware/compiler dependent functions that are provided to optimize copying of data from/to the CAN hardware buffer.

Info
CanCopyFromCan should only be used within a precopy function. CanCopyToCan should only be used within a pretransmit function.

5.6.12 Common CAN

Common CAN is a special feature which is available only on request and on systems with 2 or more CAN controllers. The idea of this feature is to map different HW channels into one application channel.

When Common CAN is activated additional receive FullCAN messages can be configured on a channel. This is realized by using a second CAN controller for the same channel. The first CAN controller (CAN A) supports Tx, Rx Full CAN and Rx Basic CAN. The second CAN controller (CAN B) supports Rx Full CAN. Both CAN controllers have to be connected to the same CAN bus. The API is always ‘Multiple Receive Channel’.

To enable the Common CAN feature activate the corresponding checkbox in the channel settings.

First select the messages handled in Full CAN objects. Then select the “Hardware Channel” to be used to receive the full CAN message.

Please note that the messages received on CAN B of the Common CAN must be filtered out with the Basic CAN mask.

5.6.13 Multiple ECU

The feature Multiple ECU is usually used for nodes that exist more than once in a car with only a few differences. At power up the application decides which node should be realized, e.g. left passenger door, or right driver door.

To reduce the memory consumption messages that are sent exclusively from one node can be overlapped with the exclusively sent messages from the other nodes. The result of this overlapping is that all these messages share a common memory location for the transmit data.

5.6.14 Signal Access Macros

Signal access macros are function like macros, to access signals within a message. They can be used by the application for an easy access to signals. The generation of signal value access macros can be enabled or disabled. If enabled, the Generation Tool generates access macros using the signal names from the communication data base with respect to prefixes or post-fixes defined in the Names tab.

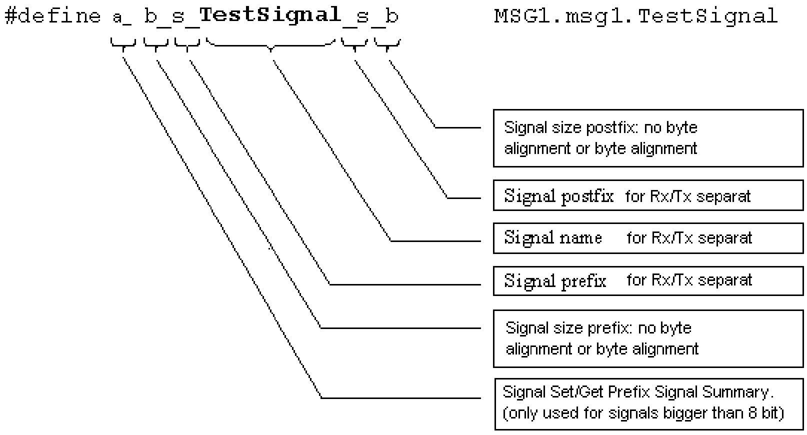

#define a_ b_s_TestSignal_s_b
MSG1.msg1.TestSignal
Signal size postfix: no byte alignment or byte alignment
Signal postfix for Rx/Tx separat
Signal name for Rx/Tx separat
Signal prefix for Rx/Tx separat
Signal size prefix: no byte alignment or byte alignment
Signal Set/Get Prefix Signal Summary. (only used for signals bigger than 8 bit)

Figure 5-13 Name of signal access macros

For each signal an access macro is formed from the signal name in the CAN database, a signal variable prefix (access via signal structures or byte/word commands), a signal prefix, a signal postfix, and a signal variable postfix. Prefixes and postfixes can be configured by the user in the generator program. To assure better readability, it is advisable not to use all four prefixes and postfixes simultaneously.

The access macros for the CAN receive buffer get an extended prefix `CAN_`. Within Precopy and Pretransmit routines these macros serve to access the CAN controller's CAN receive and transmit buffer on a signal basis.

5.6.15 CAN RAM Check

The CAN driver supports a check of the CAN controller’s mailboxes. The CAN controller RAM check is called internally every time the function CanInit() is called. The CAN driver verifies that no used mailboxes is corrupt. A mailbox is considered corrupt if a predefined pattern is written to the appropriate mailbox registers and the read operation does not return the expected pattern. If a corrupt mailbox is found the function ApplCanCorruptMailbox() is called. This function tells the application which mailbox is corrupt.

After the check of all mailboxes the CAN driver calls the callback function ApplCanMemCheckFailed() if at least one corrupt mailbox was found. The application must decide if the CAN driver disables communication or not by means of the callback function’s return value. If the application has decided to disable the communication there is no possibility to enable the communication again until the next call to CanInitPowerOn().

The CAN RAM check functionality itself can be activated via Generation Tool. This include the callback ApplCanMemCheckFailed(). The callback ApplCanCorruptMailbox() can only be activated via a user configuration file.

6 Detailed Description of the Functional Scope (High End extension)

6.1 Transmission

6.1.1 Low-Level Message Transmit

Using a multiple channel CAN Driver the routing of complete CAN messages from one CAN Bus to another one is supported by the function

```
vuint8 CanMsgTransmit(...);
```

This function has a parameter with a pointer to a CAN Message Buffer. So it is possible to route the whole buffer from one CAN chip to the other one. To prevent a conflict with the functional messages, this function uses an own send buffer (If an additional buffer is available in the CAN Controller).

A special confirmation function and an initialization callback function are called.

```
void ApplCanMsgTransmitConf(...); within confirmation interrupt
void ApplCanMsgTransmitInit(...); within CanInit
```

These functions can be used by the application to implement a data queue mechanism. There is no internal transmit queue for this transmit object available.

CanMsgTransmit() can also be used for dynamic transmission. Therefore the CAN driver supports macros to write standard ID or extended ID, DLC and data to the structure:

```
CanMsgTransmitSetStdId (...)
CanMsgTransmitSetExtId (...)
CanMsgTransmitSetDlc (...)
CanMsgTransmitSetData (...)
```

6.2 Reception

6.2.1 Multiple Basic CAN

To improve efficiency of the hardware filtering and reduce the interrupt load produced by reception of unwanted messages, the number of Hardware Basic CAN objects can be changed in the Generation Tool. Each Hardware Basic CAN object has it’s own filter.

Increasing the number of Basic CAN objects will reduce the number of available Full CAN objects (Rx and Tx).

This feature is only available for Full CAN controllers.

6.2.2 Rx Queue

The Rx Queue is a data queue which stores receive messages if the application does not want to process them within the interrupt context. In some applications it may happen that the run time in the receive interrupt becomes too long. This leads to long interrupt latencies and possible loss of messages. The Rx Queue may help in these cases. If the Rx Queue is configured the received messages are copied into this queue in the interrupt context. The handling of the queued messages is done on task level. Messages which are received by means of a RangePrecopy can also be copied into the queue or handled in the interrupt context.

In order to handle the queued messages the application has to call cyclically a CAN Driver function which checks for queued messages and processes them.

If Precopy and Indication functions are used for application messages, be aware that they are not called in interrupt context any more.

By using the Rx Queue the runtime in the Rx interrupt is decreased. The average runtime of the application is increased because of the overhead for handling the queue.

⚠	**Please note** In case a range is configured to be handled via the Rx Queue, the return code of the RangePrecopy for this range is ignored.

6.2.2.1 Handling in Receive Interrupt

During the receive interrupt the CAN Driver calls the callback function ApplCanPreRxQueue() after the range or search algorithm match in order to let the application decide whether the received message is processed within the interrupt or has to be entered into the Rx queue. The function ApplCanPreRxQueue() is only called if configured. Otherwise all received messages are handled by the Rx queue. If the Rx Queue is full the CAN Driver notifies the application by calling the callback function ApplCanRxQueueOverrun() (if configured) and discards the received message. After the message was copied into the Rx queue the Rx interrupt is terminated.

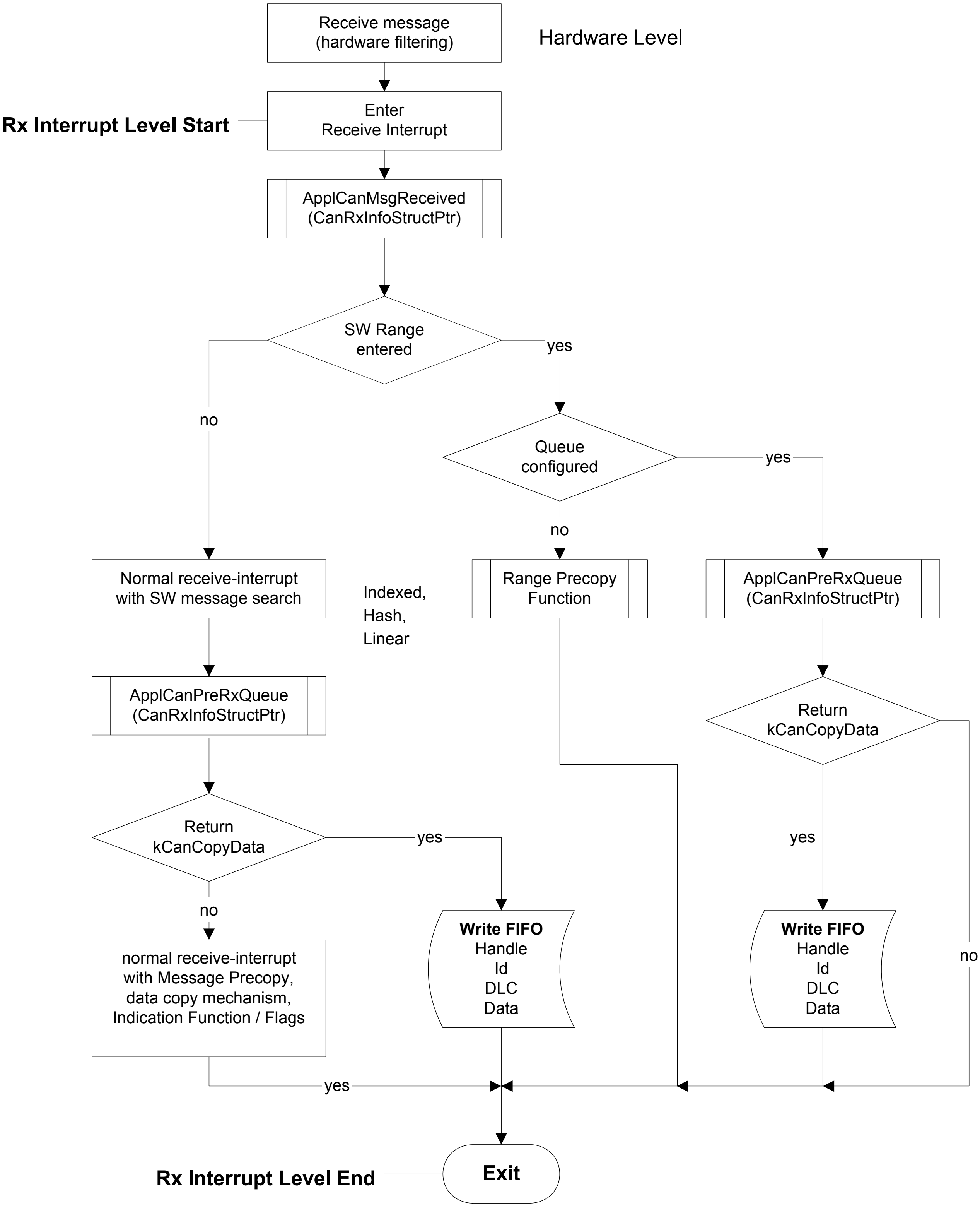


Figure 6-1 Handling of the Rx queue within the receive routine.

6.2.2.2 Handling on Task Level

In order to process the messages pending in the Rx queue the application has to call the function CanHandleRxMsg(). This function processes all messages in the queue. The processing of the messages is done in the same way like in the Rx interrupt. That means for each message the Generic Precopy and the UserPrecopy are called. After that the message data are copied into the RAM buffer and than the Indication Flag is set and the UserIndication function is called. The last step is to delete the processed message from the Rx queue.

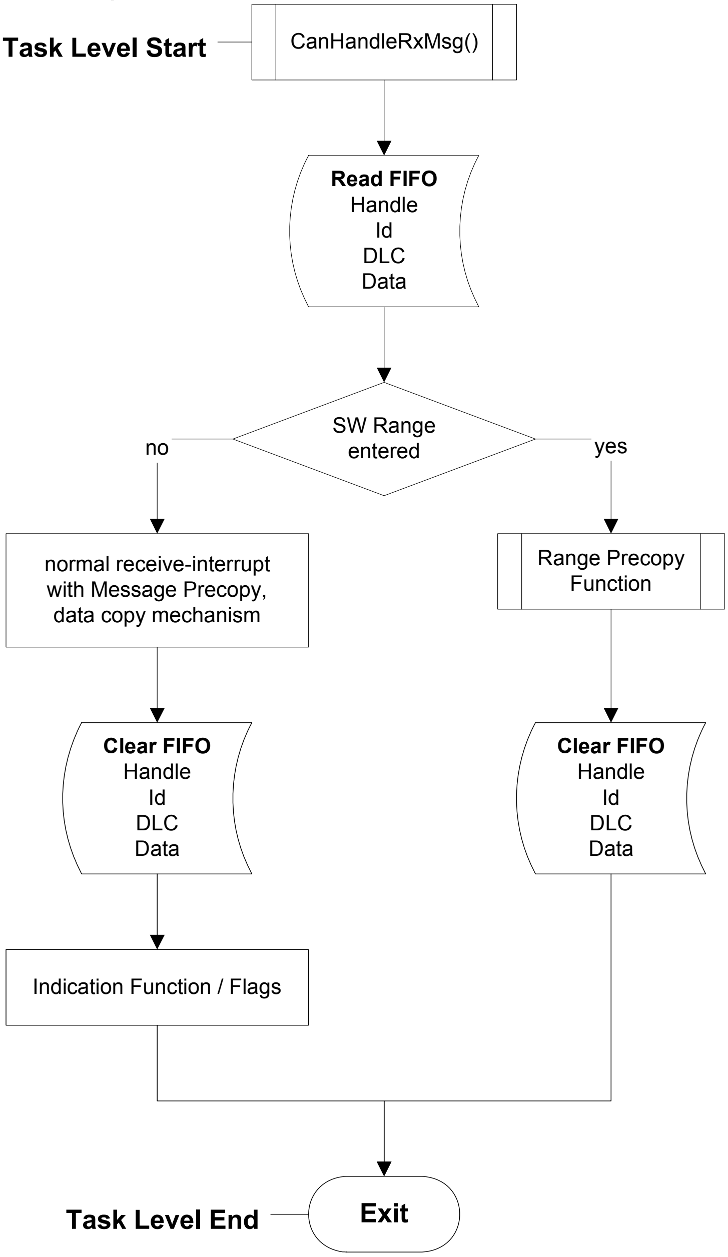


Figure 6-2 Handling of the Rx queue on task level.

6.2.2.3 Resetting the Rx Queue

The CAN Driver provides the function CanDeleteRxQueue() to delete all messages pending in the Rx Queue

6.3 Special Features

6.3.1 Individual Polling

Each mailbox (BasicCAN Rx, FullCAN Rx, FullCAN Tx, low level Tx and normal Tx) can be selected to be polled or treat in interrupt context. This also provides the possibility to use interrupt mode on one channel and polling mode on the other.

The polling tasks of the standard polling mode are still available. The CAN Driver provides additional service functions to poll each mailbox individual.

```
void CanRxBasicCANObjTask ( ... );
void CanRxFullCANObjTask  ( ... );
void CanTxObjTask       ( ... );
```

These functions have the number of the mailbox and the hardware channel as parameter. For both parameters, symbolic names are generated.

`CanTask()`, `CanErrorTask()` and `CanWakeUpTask()` are available in this polling mode, too.

7 Feature List (Standard and High End)

This general feature list describes the overall feature set of Vector CAN Drivers. Not all of these features are mandatory for all CAN Drivers. Please refer to the CAN Controller dependent CAN Driver manual for details TechnicalReference_CAN_<hardware>.pdf [#hw_feature].

CAN Driver Functionality

		Standard	HighEnd	Functions
Initialization				
Power-On Initialization		■	■	CanInitPowerOn
Re-Initialization		■	■	CanInit
Transmission				
Transmit Request		■	■	CanTransmit
Transmit Request Queue		■	■	
Internal data copy mechanism		■	■	
Pretransmit functions		■	■	UserPreTransmit
Common confirmation function		■	■	ApplCanTxConfirmation
Confirmation flag		■	■	
Confirmation function		■	■	UserConfirmation
Offline Mode		■	■	CanOnline, CanOffline
Partial Offline Mode		■	■	CanOnline, CanPartOffline, CanGetPartMode
Passive-Mode		■	■	CanSetActive, CanSetPassive
Tx Observe mode		■	■	ApplCanInit, ApplCanTxObjStart, ApplCanTxObjConfirmed
Dynamic TxObjects	ID	■	■	CanDynTxSet(Ext)Id
	DLC	■	■	CanDynTxSetDlc
	Data-Ptr	■	■	CanDynTxSetDataPtr
Full CAN Tx Objects		□	□	
Cancellation in Hardware		□	□	CanCancelTransmit, CanCancelMsgTransmit
Low Level Message Transmit			■	CanMsgTransmit
Reception				
Receive function		■	■	ApplCanMsgReceived
Search algorithms	Linear	■	■	
	Table			
	Index		■	
	Hash	■	■	

Range specific precopy functions		4	4	ApplCanRangeXxxPrecopy Xxx .. 0,1,2,3
DLC check		■	■	ApplCanMsgDlcFailed
Internal data copy mechanism		■	■	
Generic precopy function		■	■	ApplCanGenericPrecopy
Precopy function		■	■	UserPrecopy
Indication flag		■	■	
Indication function		■	■	UserIndication
Message not matched function		■	■	ApplCanMsgNotMatched
Overrun Notification		■	■	ApplCanOverrun
Full-CAN overrun notification		□	□	ApplCanFullCanOverrun
Multiple Basic CAN			■	
Rx Queue			■	CanHandleRxMsg, CanDeleteRxQueue
Bus off				
Notification function		■	■	ApplCanBusOff
Nested Recovery functions		■	■	CanResetBusOffStart, CanResetBusOffEnd
Sleep Mode				
Mode Change		□	□	CanSleep, CanWakeUp
Preparation		■	■	CanResetBusSleep
Notification function		□	□	ApplCanWakeUp
Special Feature				
Status		■	■	CanGetStatus
Security Level		■	■	
Assertions		■	■	ApplCanFatalError
Hardware loop check		■	■	ApplCanTimerStart ApplCanTimerLoop ApplCanTimerEnd
Stop Mode		□	□	CanStart, CanStop
Support of OSEK operating system		■	■	
Standard Polling Mode	Tx	■	■	CanTxTask
	Rx(Full CAN objects)	□	□	CanRxFullCANTask
	Rx(Basic CAN objects)	■	■	CanRxBasicCANTask
	Error	■	■	CanErrorTask
	Wakeup	□	□	CanWakeUpTask
Individual Polling			■	CanTxObjTask, CanRxFullCANObjTask, CanRxBasicCANObjTask
Multi-channel		■	■	
Support extended ID addressing mode		■	■	
Support mixed ID addressing mode		■	■	
Support access to error counters		■	■	CanRxActualErrorCounter CanTxActualErrorCounter

Copy functions	■	■	CanCopyFromCan, CanCopyToCan
CAN RAM check	■	■	ApplCanMemCheckFailed

Figure 2: Feature List

■ feature is supported in general (exceptions might be possible if a CAN controller is not able to support a feature.

□ feature is not implemented for each hardware because different CAN controller doesn’t support this feature.

8 Description of the API (Standard)

The complete Standard CAN Driver API is described in this section.

8.1 API Categories

Depending on the number of supported channels, i.e. the number of connected CAN networks to one ECU, the API of the CAN Driver is realized as "Single Channel" or "Multiple Channel" with additional channel specific information.

8.1.1 Single Receive Channel (SRC)

A “Single Receive Channel” CAN Driver supports one CAN channel.

8.1.2 Multiple Receive Channel (MRC)

A "Single Receive Channel" CAN Driver is typically extended for multiple channels by adding an index to the function parameter list (e.g. CanOnline() becomes to CanOnline(channel)) or by using the handle as a channel indicator (e.g. CanTransmit(txHandle)).

8.2 Data Types

The following general data types are defined by the CAN Driver:

vbittype	canbittype	Single bit information
vuint8	canuint8	unsigned 8 bit (byte) value
vuint16	canuint16	unsigned 16 bit (word) value
vuint32	canuint32	unsigned 32 bit (dword) value
vsint8	cansint8	signed 8 bit (byte) value
vsint16	cansint16	signed 16 bit (word) value
vsint32	cansint32	signed 32 bit (dword) value

There are special data types to reference specific generated data structures:

CanInitHandle	Initialization parameters
CanReceiveHandle	Receive parameters
CanTransmitHandle	Transmit parameters
CanChannelHandle	Channel parameters (only available in indexed CAN Drivers)

Some data types are referencing the CAN Controller registers

CanChipDataPtr	Receive and transmit data register of the CAN Controller
CanChipMsgPtr	Complete receive and transmit message objects including CAN identifier and DLC

Some data types are only available in Single Receive Channel and Multiple Receive Channel CAN Drivers

```
typedef volatile struct
{
  CanChipDataPtr     pChipData;
  CanTransmitHandle  Handle;
} CanTxInfoStruct;
```

Structure with transmit information.

pChipData is the pointer to the transmit data bytes in the CAN controller.

Handle of the transmit message.

```
typedef volatile struct
{
  CanChannelHandle  Channel;
  CanChipMsgPtr     pChipMsgObj;
  CanChipDataPtr    pChipData;
  CanReceiveHandle  Handle;
} tCanRxInfoStruct;
```

Structure with receive information:

Channel from which the precopy is called.

pChipMsgObj is the pointer to the CAN Controller Receive Register. If there are several receive objects with different memory addresses available, pChipMsgObj contains the pointer to the dedicated receive object to get some information like CAN identifier or DLC of the received message directly out of the CAN Controller registers.

pChipData is the pointer to the received data bytes.

	Handle of the received message.
CanRxInfoStructPtr	Pointer to structure with receive information.

8.3 Constants

This information is stored in ROM.

8.3.1 Version Number

kCanMainVersion and kCanSubVersion contains the BCD coded version of the CAN Driver:

`kCanMainVersion`	Main version of the CAN Driver (BCD coded in a vuint8 constant variable)
`kCanSubVersion`	Sub version of the CAN Driver (BCD coded in a vuint8 constant variable)
`kCanBugFixVersion`	Release version of the CAN Driver (BCD coded in a vuint8 constant variable)

Example
A version number 2.31.00 is coded as 0x02 in kCanMainVersion, 0x31 in kCanSubVersion and 0x00 in kCanBugFixVersion.

8.4 Macros

8.4.1 Conversion between Logical and Hardware Representation of CAN Parameter DLC

These macros are used to convert the CAN protocol specific parameter DLC between the logical presentation (DLC: 0..8) and the CAN Controller dependent, internal register layout of different CAN Controllers.

They are normally used by the Generation Tool for the initialization of the node specific control structures but they are available also for the Application, if necessary.

The MK_... macros are converting from the logical to the CAN Controller dependent representation:

`MK_TX_DLC(dlc)`	Conversion of transmit DLC, if associated CAN message has standard identifier

The following macro is only allowed to be used if extended CAN identifiers are used:

`MK_TX_DLC_EXT(dlc)`	Conversion of transmit DLC if associated CAN message has an extended identifier

The XT_... macro is converting from the CAN Controller dependent to the logical presentation:

`XT_TX_DLC(dlc)`	Conversion of transmit DLC independent of identifier type

8.4.2 Direct Access to the CAN Controller Registers

These macros are defined by the CAN Driver to provide the access on CAN protocol specific parameters like CAN identifier and DLC currently available in the CAN Controller.

To assign these information to a previously received message they are only valid in the callback function ApplCanMsgReceived() or in user specific precopy functions. Only in this scope there is a clear reference on receive messages possible and data in the CAN Controller receive registers are still locked. They are referencing either on the CAN Controller register or on the software shadow buffer of the CAN Driver, if used. The parameter rxStruct is only available for Single Receive Channel and Multiple Receive Channel Drivers and is the pointer to the receive information structure.

`CanRxActualId(rxStruct)`	Read identifier in logical presentation (0h..7FFh for standard identifier or 0h..1FFFFFFFh for extended identifier). In case of mixed identifier, the macro CanRxActualIdType can be used to decide whether the CAN identifier is in standard or extended format.
`CanRxActualIdType(rxStruct)`	Read the format type of the CAN identifier kCanIdTypeStd standard format kCanIdTypeExt extended format
`CanRxActualDLC(rxStruct)`	Read DLC in logical presentation (0..8)
`CanRxActualData(rxStruct,i)`	Read Data in logical presentation. i is the position of the byte (0..7).
`CanRxActualErrorCounter(channel)`	Read current status of the receive error counter. In use of microcontrollers without an readable error counter, this macro returns always 0.
`CanTxActualErrorCounter(channel)`	Read current status of the transmit error counter. In use of microcontrollers without an readable error counter, this macro returns always 0.

The following macros are only available if extended CAN identifiers are used:

`CanRxActualIdExtHi(rxStruct)`	Read the bit 24 to 29 of the extended identifier in logical presentation
`CanRxActualIdExtMidHi(rxStruct)`	Read the bit 16 to 23 of the extended identifier in logical presentation
`CanRxActualIdExtMidLo(rxStruct)`	Read the bit 8 to 15 of the extended identifier in logical presentation
`CanRxActualIdExtLo(rxStruct)`	Read the bit 0 to 7 of the extended identifier in logical presentation

To write CAN protocol specific parameters like CAN identifier and DLC the to the CAN controller there are some macros available. The parameter txStruct is only available for

Single Receive Channel and Multiple Receive Channel Drivers and is the pointer to the transmit information structure.

`CanTxWriteActId(txStruct, id)`	Write the parameter id in standard format and in logical presentation to the hardware.
`CanTxWriteActDLC(rxStruct, dlc)`	Write the DLC in logical presentation (0..8)

The following macro is only available if extended CAN identifiers are used:

`CanTxWriteActExtId( txStruct,id)`	Write the parameter id in extended format and in logical presentation to the hardware.

8.4.3 Interpretation of the CAN Status

The following macros are used to decode the return code of CanGetStatus() (TRUE means not equal to zero):

`CanHwIsOk(state)`	This macro returns TRUE, if the status of the CAN Controller is Error-Active.
`CanHwIsWarning(state)`	This macro returns TRUE, if the status of the CAN Controller is Warning (at least one error counter is equal or higher than 96).
`CanHwIsPassive(state)`	This macro returns TRUE, if the status of the CAN Controller is Error-Passive.
`CanHwIsBusOff(state)`	This macro returns TRUE, if the status of the CAN Controller is Bus-Off. This information is only temporary. The time when this status changes from TRUE to FALSE depends on the CAN controller. This could be after the CAN controller has resynchronized on the bus regardless of the Busoff recovery by the Application. This could also be after CanResetBusoffStart() is called or after CanResetBusoffEnd() is called. Hint: Busoff detection has to be performed with ApplCanBusoff().
`CanHwIsWakeup(state)`	This macro returns TRUE, if the CAN Controller is not in sleep mode.
`CanHwIsSleep(state)`	This macro returns TRUE, if the CAN Controller is in sleep mode.
`CanHwIsStart(state)`	This macro returns TRUE, if the CAN Controller is not in stop mode.
`CanHwIsStop(state)`	This macro returns TRUE, if the CAN Controller is in stop mode.
`CanIsOnline(state)`	This macro returns TRUE, if the CAN Driver is online.
`CanIsOffline(state)`	This macro returns TRUE, if the CAN Driver is offline.

Not all CAN Controllers support all of the hardware dependent states. Please refer to the CAN Controller specific documentation TechnicalReference_CAN_<hardware>.pdf [#hw_status] for details.

8.4.4 Access to low level transmit structure

These macros are defined by the CAN driver to fill the set the data to be transmitted with `CanMsgTransmit():`

`CanMsgTransmitSetStdId( tCanMsgTransmitStruct *txData, vuint16 id)`	Write the parameter id in standard format and in logical presentation to the structure txData.
`CanMsgTransmitSetExtId( tCanMsgTransmitStruct *txData, vuint32 id)`	Write the parameter id in extended format and in logical presentation to the structure txData.
`CanMsgTransmitSetDlc( tCanMsgTransmitStruct *txData, vuint8 dlc)`	Write the DLC in logical presentation (0..8)
`CanMsgTransmitSetData( tCanMsgTransmitStruct *txData, vuint8 nrDataByte, vuint8 *txDataBytes)`	Write the data bytes to be transmitted to the structure txData. nrDataBytes specifies the number of bytes to be copied (e.g. same as the DLC, max. 8). txDataBytes points to the current location where the data has to be copied from.

8.5 Functions

This chapter contains a description of the CAN Driver functions (services, callbacks and user specifics) and the appropriate parameters and return codes. The function declarations are given in C syntax as explained below:

```
vuint8 CanTransmit( CanTransmitHandle txObject );
```

- vuint8 is the type of the return code
- CanTransmit is the name of the function
- CanTransmitHandle is the type of the function parameter
- txObject is the function parameter.

8.5.1 Service Functions

8.5.1.1 CanInitPowerOn

CanInitPowerOn

Prototype	
Single Receive Channel	`void` **`CanInitPowerOn`** `( void )`
Multiple Receive Channel	
Parameter	
-	-
Return code	
-	-
Functional Description	
The service function `CanInitPowerOn()` initializes the CAN Controller and the CAN Drivers internal variables. The CAN Driver is always set to online mode and active operating state.	
Particularities and Limitations	
■ This service function has to be called before any other CAN Driver function. The interrupts have to be disabled during this service function is called. ■ For indexed CAN Drivers every channel is initialized with kCanInitObj0.	

8.5.1.2 CanInit

CanInit

Prototype	
Single Receive Channel	`void` **`CanInit`** `( CanInitHandle initObject )`
Multiple Receive Channel	`void` **`CanInit`** `( CanChannelHandle channel, CanInitHandle initObject )`
Parameter	
`initObject`	Handle of an initialization structure. The generated macros should be used: `kCanInitObjX` (with X = 1 ... Number of generated initialization structures)
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
-	-
Functional Description	
Initialization of the CAN Controller hardware. It is used to cancel pending transmit requests in the CAN Controller transmit register and to change the baud rate or the hardware acceptance filters. Online/Offline mode and Active/Passive state will not be changed.	
Particularities and Limitations	

- During the call of `CanInit(..)`, the CAN Driver has to be in offline mode.
- `CanInit(..)` is not reentrant and therefore must not be called recursively.
- `CanInit(..)` must not be interrupted by `CanReset...(..)`, `CanSleep(..)`, `CanWakeUp(..)` or by any CAN interrupt service routine and vice versa.
- `CanInit(..)` must not interrupt the confirmation interrupt and must not be called in the confirmation or indication function.

8.5.1.3 CanTransmit

CanTransmit

Prototype	
Single Receive Channel	`vuint8 **CanTransmit** ( CanTransmitHandle txObject )`
Multiple Receive Channel	
Parameter	
`txObject`	Handle of the transmit object
Return code	
`kCanTxOk`	The transmit request was accepted by the CAN Driver
`kCanTxFailed`	Error code because one of the following conditions: ■ Transmit request could not be passed to the CAN Controller because the transmit registers are busy (only if there is no transmit queue used) ■ CAN Driver's transmit path is in offline mode ■ Special hardware conditions of the CAN Controller (e.g. the sleep mode was entered; failed synchronization on the CAN bus)
`kCanTxPartOffline`	Error code because the transmit path of the CAN driver is in partial offline mode for this transmit object.

- CanTransmit(..) supports the Network Management which can enable or disable the CAN Driver's transmit path by means of the CAN Driver service functions CanOnline() and CanOffline(). No distinction is made between Network Management and Application messages. In the offline mode, the transmit request is rejected with an error code.
- For CAN Controllers with priority controlled transmit queue (hardware or software) the sequence of transmission may deviate from the call sequence of CanTransmit(..) because the transmit queues are handled according to priorities (lowest CAN identifier first) and not according to the chronological order of the entries in the queue (FIFO).
- The generated handles should be used to reference the transmit objects. The names consist of the message symbol, a prefix and a postfix. Fixed rules are used to build these names. For more details please refer to the user manual of the Generation Tool

8.5.1.4 CanTask

CanTask

Prototype	
Single Receive Channel	`void` **`CanTask`** `( void )`
Multiple Receive Channel	
Parameter	
–	-
Return code	
–	-
Functional Description	
The service function CanTask() does polling of error events, receive objects, transmit objects and wake-up events in the CAN Controller according to the configured polling mode. In multiple channel drivers the CanTask() handles all channels.	
Particularities and Limitations	
- CanTask() must not run on higher priority than other CAN functions. - CanTask() is available, if any polling mode is configured for the CAN Driver - CanTask() is also available for some CAN Controllers if cancellation in hardware is configured. See more about that in the CAN Controller specific documentation TechnicalReference_CAN_<hardware>.pdf [#hw_cancel] .	

8.5.1.5 CanTxTask

CanTxTask

Prototype	
Single Receive Channel	`void` **`CanTxTask`** `( void )`
Multiple Receive Channel	`void` **`CanTxTask`** `( CanChannelHandle channel )`
Parameter	
channel	Handle of a CAN channel. The generated macros should be used: kCanIndexX (with X = 0 ... Number of generated channel index)
Return code	
–	-

Functional Description
The service function `CanTxTask()` does polling of transmit objects in the CAN Controller. Confirmation functions will be called and confirmation flags will be set. If the transmit queue is configured, this service function additionally transmits the queued messages.
Particularities and Limitations
■ `CanTxTask()` is available, if the general polling mode or the transmit polling mode is configured. ■ `CanTxTask()` is also available for some CAN Controllers if cancellation in hardware is configured. See more about that in the CAN Controller specific documentation TechnicalReference_CAN_<hardware>.pdf [#hw_cancel] .

8.5.1.6 CanRxFullCANTask

CanRxFullCanTask

Prototype	
Single Receive Channel	`void` **`CanRxFullCANTask`** `( void )`
Multiple Receive Channel	`void` **`CanRxFullCANTask`** `( CanChannelHandle channel )`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
The service function `CanRxFullCanTask()` does polling of Full CAN receive objects (if available) according to the configured polling mode.	
Particularities and Limitations	
■ `CanRxFullCanTask()` must not run on higher priority than other CAN functions. ■ `CanRxFullCanTask()` is available if the Full CAN receive polling mode is configured.	

8.5.1.7 CanRxBasicCANTask

CanRxBasicCANTask

Prototype	
Single Receive Channel	`void` **`CanRxBasicCANTask`** `( void )`
Multiple Receive Channel	`void` **`CanRxBasicCANTask`** `( CanChannelHandle channel )`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
The service function `CanRxBasicCANTask()` does polling of Basic CAN receive objects according to the configured polling mode.	

8.5.1.10 CanOnline

CanOnline

Prototype	
Single Receive Channel	`void CanOnline ( void )`
Multiple Receive Channel	`void CanOnline ( CanChannelHandle channel )`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
The service function `CanOnline()` enables the CAN Driver's transmit path for all subsequent transmit requests of `CanTransmit(..)`. This is prerequisite to transmit any CAN message. The current status of the transmit path can be queried by `CanGetStatus()`. For indexed CAN Driver, this functionality is related to the specified CAN channel.	
Particularities and Limitations	
If a Network Management is used, the service function `CanOnline()` may only be used by the Network Management. It is only allowed to call `CanOnline()` on Task level. No other CAN Driver service function is allowed to be interrupted by `CanOnline()`.	

8.5.1.11 CanOffline

CanOffline

Prototype	
Single Receive Channel	`void CanOffline ( void )`
Multiple Receive Channel	`void CanOffline ( CanChannelHandle channel )`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
The service function `CanOffline()` disables the CAN Driver's transmit path for all subsequent transmit requests of `CanTransmit(..)`. While the transmit path is blocked, transmit requests by `CanTransmit(..)` are rejected with an error. This can be determined by evaluating the return code. The current status of the transmit path can be queried `by CanGetStatus()`. For indexed CAN Driver, this functionality is related to the specified CAN channel.	
Particularities and Limitations	

If the CAN Driver is configured to use a transmit queue, all queue entries will be cleared, i.e. transmit requests will be lost. If a Network Management is used, the service functions `CanOffline()` may only be used by the Network Management.

8.5.1.12 CanPartOnline

CanPartOnline

Prototype	
Single Receive Channel	`void` **`CanPartOnline`** `( vuint8 sendGroup )`
Multiple Receive Channel	`void` **`CanPartOnline`** `( CanChannelHandle channel, vuint8 sendGroup )`
Parameter	
`sendGroup`	Send group or groups to be switched to online.
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
The service function `CanPartOnline()` enables the CAN Driver's transmit path for the selected send groups. The current status of the partial offline mode can be queried by `CanGetPartMode()`. For indexed CAN Driver, this functionality is related to the specified CAN channel.	
Particularities and Limitations	

8.5.1.13 CanPartOffline

CanPartOffline

Prototype	
Single Receive Channel	`void` **`CanPartOffline`** `( vuint8 sendGroup )`
Multiple Receive Channel	`void` **`CanPartOffline`** `( CanChannelHandle channel, vuint8 sendGroup )`
Parameter	
`sendGroup`	Send group or groups to be switched to offline.
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	

The service function `CanPartOffline()` disables the CAN Driver's transmit path for the selected send groups. While the transmit path is blocked for a selected group, transmit requests of a message assigned to this group by `CanTransmit(..)` are rejected with `kCanPartOffline`. This can be determined by evaluating the return code. The current status of the partial offline mode can be queried by `CanGetPartMode()`. For indexed CAN Driver, this functionality is related to the specified CAN channel.
Particularities and Limitations
■ A queued message will be still send after the function `CanPartOffline()` was called.

8.5.1.14 CanGetPartMode

CanGetPartMode

Prototype	
Single Receive Channel	`vuint8 **CanGetPartMode** ( void )`
Multiple Receive Channel	`vuint8 **CanGetPartMode** ( CanChannelHandle channel )`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
`vuint8`	Send groups which are in partial offline mode. ■ C_SEND_GRP_NONE (if the partial mode is inactive) ■ C_SEND_GRP_ALL (if the partial mode is active for all groups.) NOTE: predefined macros can be used to check for all or none send groups. For indexed CAN Driver, this functionality is related to the specified CAN channel. See also 5.2.6 Partial Offline Mode page 35
Functional Description	
Reads the current partial offline status of the CAN Driver.	
Particularities and Limitations	

8.5.1.15 CanGetStatus

CanGetStatus

Prototype	
Single Receive Channel	`vuint8 **CanGetStatus** ( void )`
Multiple Receive Channel	`vuint8 **CanGetStatus** ( CanChannelHandle channel )`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)

Return code	
`vuint8`	Software status of the CAN Driver. The following information is coded in the return code: ■ CAN Driver is offline (`CanOffline()` was called) If extended status is enabled, this function also returns the hardware status of the CAN Controller. The following additional information are coded in the return code: ■ Warning level, Error-Active/-Passive state and Bus-Off of the CAN Controller ■ CAN Controller is in sleep mode (`CanSleep()` was called) ■ CAN Controller is in stop mode (`CanStop()` was called) There are special macros to get this information in the return code. These macros are `TRUE` (not equal to 0) if the specific condition is valid and `FALSE` (equal to 0) if not. The parameter of these macros is the status, i.e. the return code of `CanGetStatus()`: ■ `CanHwIsWarning(..), CanHwIsPassive(..), CanHwIsBusOff(..),` ■ `CanHwIsOk(..)` ■ `CanHwIs Sleep(..), CanHwIsWakeup(..)` ■ `CanHwIsStop(..), CanHwIsStart(..)` ■ `CanIsOffline(..), CanIsOnline(..)` For indexed CAN Driver, this functionality is related to the specified CAN channel.
Functional Description	
Reads the current status of the CAN Driver and the CAN Controller.	
Particularities and Limitations	

8.5.1.16 CanSleep

CanSleep

Prototype	
Single Receive Channel	`vuint8 CanSleep (void )`
Multiple Receive Channel	`vuint8 CanSleep ( CanChannelHandle channel )`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
Result of the sleep request:	
`kCanFailed`	Sleep mode not entered
`kCanOk`	Sleep mode entered
`kCanNotSupported`	The function `CanSleep` is not supported by this driver
Functional Description	
The service function `CanSleep()` puts the CAN Controller into sleep mode. This reduces the power	

consumption of the CAN Controller and enables the wake up behavior if the CAN Controller supports this functionality. For indexed CAN Driver, this functionality is related to the specified CAN channel.

Particularities and Limitations

- This functionality is not supported for all CAN Controllers. In such case the function is provided by the CAN Driver but without any effect on the CAN Controller. This is done to enable the Application to realize an orthogonal software structure.
- If it is supported by the CAN Controller, on a wake-up by the CAN bus, the callback function `ApplCanWakeUp()` is called.
- If a message is currently transmitted or received during the call of this service function, a direct wake-up interrupt occurs or the CAN Driver remains in this function until the sleep mode is entered. This behavior of the CAN Controller has to be considered in implementing the Application or the Network Management. (This behavior is hardware dependent and described more detailed in the CAN controller specific documentation TechnicalReference_CAN_<hardware>.pdf [#hw_sleep])
- If the sleep mode is not entered, no CAN wake-up interrupt occurs on the detection of any message on the CAN bus. The callback function `ApplCanWakeUp()` will not be called and in consequence the bus transceiver will not be initialized. This leads to CAN bus errors. Therefore it is necessary to call a set of functions to realize a wake-up capable system. The order of the function calls is very important. more…
- During the call of `CanSleep()` the CAN Driver has to be offline.
- `CanSleep()` must not be interrupted by `CanInit()`, `CanReset...()`, `CanWakeUp()` or any CAN interrupt routine and vice versa.
- `CanSleep(..)` is not reentrant and therefore must not be called recursively.
- It isn’t allowed to call `CanSleep()` out of any callback function.
- CAN Interrupts should be disabled during the call of `CanSleep()`. To disable the Can Interrupts the function `CanCanInterruptDisable()` should not be used. If this function is used no CAN wake-up interrupt occurs on the detection of any message on the CAN bus. The callback function `ApplCanWakeUp()` will not be called.
- In Sleep mode the service functions `CanGetStatus()`, `CanWakeUp()`, `CanTransmit()`, `CanTask() and all Can...Task()` are allowed to be called.

8.5.1.17 CanWakeUp

CanWakeUp

Prototype	
Single Receive Channel	`vuint8 **CanWakeUp** ( void )`
Multiple Receive Channel	`vuint8 **CanWakeUp** ( CanChannelHandle channel )`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
Result of the wakeup request	
`kCanFailed`	wakeup was not successful
`kCanOk`	Sleep mode left
`kCanNotSupported`	The function `CanWakeUp` is not supported by this driver.
Functional Description	

The service function `CanWakeUp()` enters the normal operating mode of the CAN Controller.

For indexed CAN Driver, this functionality is related to the specified CAN channel.

Particularities and Limitations

- This functionality is not supported for all CAN Controllers. In such case the function is provided by the CAN Driver but without any effect on the CAN Controller. This is done to enable the Application to realize an orthogonal software structure.
- During the call of `CanWakeUp()` the CAN Driver has to be offline.
- No wake-up interrupt is generated by the call of `CanWakeUp()`.
- `CanWakeUp()` must not be interrupted by `CanInit()`, `CanReset...()`, `CanSleep()` or any CAN interrupt routine and vice versa.
- `CanWakeUp(..)` is not reentrant and therefore must not be called recursively.

8.5.1.18 CanStart

CanStart

Prototype	
Single Receive Channel	`vuint8 **CanStart** ( void )`
Multiple Receive Channel	`vuint8 **CanStart** ( CanChannelHandle channel )`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
Result of the stop request	
`kCanFailed`	Restart of the CAN controller was not successful.
`kCanOk`	Stop mode left
`kCanNotSupported`	The function `CanStart()` is not supported by this driver.

Functional Description

The service function `CanStart()` enters the normal operating mode of the CAN Controller. `CanStart()` may not be called in sleep mode.

For indexed CAN Driver, this functionality is related to the specified CAN channel.

Particularities and Limitations

- This functionality is not supported for all CAN Controllers. In such case the function is provided by the CAN Driver but without any effect on the CAN Controller. This is done to enable the Application to realize an orthogonal software structure.
- During the call of `CanStart()` the CAN Driver has to be offline.
- `CanStart()` must not be interrupted by `CanInit()`, `CanReset...()`, `CanWakeUp()` or any CAN interrupt routine and vice versa.
- `CanStart(..)` is not reentrant and therefore must not be called recursively.

8.5.1.19 CanStop

CanStop

Prototype	
Single Receive Channel	`vuint8 **CanStop** ( void )`
Multiple Receive Channel	`vuint8 **CanStop** ( CanChannelHandle channel )`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
Result of the stop request:	
`kCanFailed`	Stop mode not entered
`kCanOk`	Stop mode entered
`kCanNotSupported`	The function `CanStop` is not supported by this driver.
Functional Description	
The service function `CanStop()` puts the CAN Controller into stop or hold mode. This does not reduces the power consumption of the CAN Controller. The stop mode can only be left by calling `CanStart().CanStop()` must not be called in sleep mode. For indexed CAN Driver, this functionality is related to the specified CAN channel.	
Particularities and Limitations	
■ This functionality is not supported for all CAN Controllers. In such case the function is provided by the CAN Driver but without any effect on the CAN Controller. This is done to enable the Application to realize an orthogonal software structure. ■ During the call of `CanStop()` the CAN Driver has to be offline. ■ `CanStop()` must not be interrupted by `CanInit()`, `CanReset...()`, `CanWakeUp()` or any CAN interrupt routine and vice versa. ■ `CanStop(..)` is not reentrant and therefore must not be called recursively.	

8.5.1.20 CanGlobalInterruptDisable

CanGlobalInterruptDisable

Prototype	
Single Receive Channel	`void **CanGlobalInterruptDisable** ( void )`
Multiple Receive Channel	
Parameter	
–	-
Return code	
–	-
Functional Description	
The service function `CanGlobalInterruptDisable()` disables interrupts, either by changing the global interrupt control flag of the microprocessor or the interrupt level of the interrupt controller. In the later case, the interrupt level is configurable. All levels where the CAN API (CAN interrupt, Flags, service functions) is used have to be disabled.	

Particularities and Limitations
■ This function has been moved to VstdLib. For more information refer to Application note-ISC-2-1050_VstdLibIntegration.pdf

8.5.1.21 CanGlobalInterruptRestore

CanGlobalInterruptRestore

Prototype	
Single Receive Channel Multiple Receive Channel	`void CanGlobalInterruptRestore ( void )`
Parameter	
–	-
Return code	
–	-
Functional Description	
The service function `CanGlobalInterruptRestore()` restores the initial interrupt state which was saved temporarily by `CanGlobalInterruptDisable()`. If `CanGlobalInterruptDisable()` is called in a nested way, the initial interrupt state is not restored until `CanGlobalInterruptRestore()` has been called as many times.	
Particularities and Limitations	
■ This function has been moved to VstdLib. For more information refer to Application note AN-ISC-2-1050_VstdLibIntegration.pdf	

8.5.1.22 CanCanInterruptDisable

CanCanInterruptDisable

Prototype	
Single Receive Channel	`void CanCanInterruptDisable ( void )`
Multiple Receive Channel	`void CanCanInterruptDisable ( CanChannelHandle channel )`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
The service function `CanCanInterruptDisable()` disables all CAN interrupts of one CAN channel, either by changing the CAN interrupt control flags of the interrupt controller or of the CAN controller. In case of separately implemented wake-up interrupt routines they have to be disabled by the application. Therefore the callback function `ApplCanAddCanInterruptDisable()` can be activated.	
Particularities and Limitations	

■ The CAN Drivers differ in the implementation of this service function. Please refer to the CAN Controller specification documentation TechnicalReference_CAN_<hardware>.pdf [#hw_int] for details. ■ This service function is not allowed to be called during Sleep-Mode.	

8.5.1.23 CanCanInterruptRestore

CanCanInterruptRestore

Prototype	
Single Receive Channel	`void` **`CanCanInterruptRestore`** `( void )`
Multiple Receive Channel	`void` **`CanCanInterruptRestore`** `( CanChannelHandle channel )`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
The service function `CanCanInterruptRestore()` restores the initial interrupt state which was saved temporarily by `CanCanInterruptDisable()`. If `CanCanInterruptDisable()` is called in a nested way, the initial interrupt state is not restored until `CanCanInterruptRestore()` has been called as many times. In case of separately implemented wake-up interrupt routines they have to be restored by the application. Therefore the callback function `ApplCanAddCanInterruptRestore()` can be activated.	
Particularities and Limitations	
■ The CAN Drivers differ in the implementation of this service function. Please refer to the CAN Controller specification documentation TechnicalReference_CAN_<hardware>.pdf [#hw_int] for details. ■ This service function is not allowed to be called during Sleep-Mode.	

8.5.1.24 CanSetPassive

CanSetPassive

Prototype	
Single Receive Channel	`void` **`CanSetPassive`** `( void )`
Multiple Receive Channel	`void` **`CanSetPassive`** `( CanChannelHandle channel )`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
The service function `CanSetPassive(..)` switches the CAN Driver to the passive state. For indexed CAN Driver, this functionality is related to the specified CAN channel.	

Particularities and Limitations
■ If the CAN Driver is configured to use a transmit queue, all queue entries will be cleared, i.e. transmit requests and subsequent confirmations will be lost. ■ The passive state of the CAN Driver will have an effect only if it is enabled by the CAN Driver configuration. Nevertheless this service function is available at any time

8.5.1.25 CanSetActive

CanSetActive

Prototype	
Single Receive Channel	void **CanSetActive** (void)
Multiple Receive Channel	void **CanSetActive** (CanChannelHandle channel)
Parameter	
channel	Handle of a CAN channel. The generated macros should be used: kCanIndexX (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
The service function CanSetActive() switches the CAN Driver back to the active state. For indexed CAN Driver, this functionality is related to the specified CAN channel.	
Particularities and Limitations	
■ The passive state of the CAN Driver will have an effect only if it is enabled by the CAN Driver configuration. Nevertheless this service function is available at any time.	

8.5.1.26 CanResetBusOffStart

CanResetBusOffStart

Prototype	
Single Receive Channel	void **CanResetBusOffStart** (CanInitHandle initObject)
Multiple Receive Channel	void **CanResetBusOffStart** (CanChannelHandle channel, CanInitHandle initObject)
Parameter	
initObject	Handle of an initialization structure. The generated macros should be used: kCanInitObjX (with X = 1 ... Number of generated initialization structures)
channel	Handle of a CAN channel. The generated macros should be used: kCanIndexX (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	

This service function starts error recovery of the CAN Controller directly after BusOff. Usually a re-initialization of the CAN Controller is done. The correct handling of a BusOff depends on the used CAN Controller. Please refer to the CAN Controller specification documentation TechnicalReference_CAN_<hardware>.pdf [#hw_busoff] for details. For indexed CAN Driver, this functionality is related to the specified CAN channel.
Particularities and Limitations
■ During the call of `CanResetBusOffStart(..)`, the CAN Driver has to be in offline mode. ■ `CanResetBusOffStart(..)` is not reentrant and therefore must not be called recursively. ■ `CanResetBusOffStart()` must not be interrupted by `CanInit()`, `CanResetBusOffEnd()`, `CanResetBusSleep()`, `CanSleep()`, `CanWakeUp()` or by any CAN interrupt service routine and vice versa. ■ This service function can be realized as a preprocessor macro.

8.5.1.27 CanResetBusOffEnd

CanResetBusOffEnd

Prototype	
Single Receive Channel	`void CanResetBusOffEnd ( CanInitHandle initObject )`
Multiple Receive Channel	`void CanResetBusOffEnd ( CanChannelHandle channel, CanInitHandle initObject )`
Parameter	
`initObject`	Handle of an initialization structure. The generated macros should be used: `kCanInitObjX` (with X = 1 ... Number of generated initialization structures)
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
-	-
Functional Description	
Completes the error recovery after BusOff. For most of the CAN Drivers this service function has no effect. For indexed CAN Driver, this functionality is related to the specified CAN channel.	
Particularities and Limitations	
■ During the call of `CanResetBusOffEnd(..)`, the CAN Driver has to be in offline mode. ■ `CanResetBusOffEnd(..)` is not reentrant and therefore must not be called recursively. ■ `CanResetBusOffEnd()` must not be interrupted by `CanInit()`, `CanResetBusOffStart()`, `CanResetBusSleep()`, `CanSleep()`, `CanWakeUp()` or by any CAN interrupt service routine and vice versa. ■ This service function can be realized as a preprocessor macro.	

8.5.1.28 CanResetBusSleep

CanResetBusSleep

Prototype

Single Receive Channel	`void` **`CanResetBusSleep`** `( CanInitHandle initObject )`

Multiple Receive Channel	`void` **`CanResetBusSleep`** `( CanChannelHandle channel,`
	`CanInitHandle initObject )`

Parameter	
`initObject`	Handle of an initialization structure. The generated macros should be used: `kCanInitObjX` (with X = 1 ... Number of generated initialization structures)
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
This service function aborts pending transmit requests in the CAN Controller before the sleep mode of the CAN Controller is entered. This can be done by different ways, depending on CAN Controller specific features: ■ Complete re-initialization of the CAN Controller (using the service function `CanInit()`) ■ Cancel of the transmit requests Please refer to the CAN Controller specific documentation for details. For indexed CAN Driver, this functionality is related to the specified CAN channel.	
Particularities and Limitations	
■ During the call of `CanResetBusSleep(..)`, the CAN Driver has to be in offline mode. ■ `CanResetBusSleep(..)` is not reentrant and therefore must not be called recursively. ■ `CanResetBusSleep()` must not be interrupted by `CanInit()`, `CanResetBusOffStart()`, `CanResetBusOffEnd()`, `CanSleep()`, `CanWakeUp()` or by any CAN interrupt service routine and vice versa. ■ This service function can be realized as a preprocessor macro.	

8.5.1.29 CanGetDynTxObj

CanGetDynTxObj

Prototype	
Single Receive Channel Multiple Receive Channel	`CanTransmitHandle` **`CanGetDynTxObj`** `( CanTransmitHandle txObject )`
Parameter	
`txObject`	Handle of the dynamic transmit object.
Return code	
`CanTransmitHandle`	Handle of the dynamic transmit object or `kCanNoTxDynObjAvailable` if no dynamic transmit object is available or the specific dynamic object is already used.
Functional Description	

Reserves a dynamic transmit object.

To use dynamic transmit objects an Application must reserve a dynamic transmit object from the CAN Driver.

Before transmission, the Application must set all configured dynamic parameters of the dynamic transmit object.

The Application can use a dynamic transmit object for one or many transmissions, but finally it must release the dynamic transmit object by calling `CanReleaseDynTxObj(..)`.

Particularities and Limitations

- This service function is only available, if dynamic transmit objects are configured.
- The generated handles should be used to reference the transmit objects. The names consist of the message symbol, a prefix and a postfix. Fixed rules are used to build these names. For more details please refer to the online help of the Generation Tool.

8.5.1.30 CanReleaseDynTxObj

CanReleaseDynTxObj

Prototype	
Single Receive Channel	`vuint8 **CanReleaseDynTxObj** ( CanTransmitHandle txObject )`
Multiple Receive Channel	
Parameter	
`txObject`	Handle of the dynamic transmit object which was returned by `CanGetDynTxObj(..)`
Return code	
`kCanDynReleased`	Dynamic object is released
`kCanDynNotReleased`	Dynamic transmit object couldn't be released because the object is still in the transmit queue or in the transmit register of the CAN Controller. `CanReleaseDynTxObj(..)` has to be called later again.
Functional Description	
Release a dynamic transmit object, which was reserved before by calling `CanGetDynTxObj(..)`. The dynamic transmit object is referenced by `txObject`. After a transmission of one or more messages is finished, the Application has to release the reserved resource, because the number of dynamic transmit objects is limited and the Application should not keep reserved dynamic transmit objects for a longer time.	
Particularities and Limitations	
■ This service function is only available, if dynamic transmit objects are configured. ■ The parameter `txObject` was reserved before by a call to `CanGetDynTxObj(..)`.	

8.5.1.31 CanDynTxObjSetId

CanDynTxObjSetId

Prototype	
Single Receive Channel	`void **CanDynTxObjSetId** ( CanTransmitHandle txObject, vuint16 id )`
Multiple Receive Channel	
Parameter	
`txObject`	Handle of the dynamic transmit object which was returned by `CanGetDynTxObj(..)`.
`id`	CAN identifier in standard format
Return code	
–	-
Functional Description	
Sets the CAN identifier in standard format of a dynamic transmit object. The dynamic transmit object is referenced by txObject.	
Particularities and Limitations	
■ This service function is only available, if dynamic transmit objects are configured. ■ The parameter txObject was reserved before by a call `to CanGetDynTxObj(..).`	

8.5.1.32 CanDynTxObjSetExtId

CanDynObjSetExtId

Prototype	
Single Receive Channel Multiple Receive Channel	`void CanDynTxObjSetExtId ( CanTransmitHandle txObject, vuint16 idExtHi, vuint16 idExtLo)`
Parameter	
`txObject`	Handle of the dynamic transmit object which was returned by `CanGetDynTxObj(..)`
`idExtHi`	Upper 16 bit of the CAN identifier in extended format
`idExtLo`	Lower 16 bit of the CAN identifier in extended format
Return code	
–	-
Functional Description	
Sets the CAN identifier in extended format of a dynamic transmit object. The dynamic transmit object is referenced by txObject	
Particularities and Limitations	
■ This service function is only available, if dynamic transmit objects are configured. ■ The parameter txObject was reserved before by a call to `CanGetDynTxObj(..)`.	

8.5.1.33 CanDynTxObjSetDlc

CanDynTxObjSetDlc

Prototype	
Single Receive Channel Multiple Receive Channel	`void CanDynTxObjSetDlc ( CanTransmitHandle txObject, vuint8 dlc )`
Parameter	
`txObject`	Handle of the dynamic transmit object which was returned by `CanGetDynTxObj(..)`
`dlc`	Data Length Code of the dynamic transmit object
Return code	
–	-
Functional Description	
Sets the Data Length Code of a dynamic transmit object. The dynamic transmit object is referenced by txObject.	
Particularities and Limitations	
■ This service function is only available, if dynamic transmit objects are configured. ■ The parameter txObject was reserved before by a call to `CanGetDynTxObj(..)`.	

8.5.1.34 CanDynTxObjSetDataPtr

CanDynTxObjSetDataPtr

Prototype	
Single Receive Channel	`void` **`CanDynTxObjSetDataPtr`** `( CanTransmitHandle txObject, vuint8 *pData )`
Multiple Receive Channel	
Parameter	
`txObject`	Handle of the dynamic transmit object which was returned by `CanGetDynTxObj(..)`
`*pData`	Data reference of the application specific data buffer referenced by the dynamic transmit object
Return code	
–	-
Functional Description	
Sets the data pointer of a dynamic transmit object. The dynamic transmit object is referenced by txObject.	
Particularities and Limitations	
■ This service function is only available, if dynamic transmit objects are configured. ■ The parameter txObject was reserved before by a call to `CanGetDynTxObj(..)`.	

8.5.1.35 CanCancelTransmit

CanCancelTransmit

Prototype	
Single Receive Channel	`void` **`CanCancelTransmit`** `( CanTransmitHandle txObject )`
Multiple Receive Channel	
Parameter	
`txObject`	Handle of the transmit object
Return code	
–	-
Functional Description	
The call of the confirmation function resp. setting of the confirmation flag associated with txObject are suppressed, if this message is already in the transmit buffer of the CAN controller. If the transmit queue is enabled, a pending transmit request in the queue is canceled.	
Particularities and Limitations	
The function call of `CanCancelTransmit()` must not interrupt the transmit ISR, `CanTransmit()` or the `CanTxTask()`. Though a transmission is canceled it will be sent if the request has been already in the hardware object. Only if activated and highly dependent on hardware and vehicle manufacturer the transmit request can be deleted in the hardware transmit object, too.	

8.5.1.36 CanCopyFromCan

CanCopyFromCan

Prototype	
Single Receive Channel	`void CanCopyFromCan (void *dst, CanChipDataPtr src, vuint8 len)`
Multiple Receive Channel	
Parameter	
`dst`	Pointer to the destination in default memory. This pointer is available in the Precopy Function.
`src`	Pointer to the source CAN buffer or temporary buffer
`len`	number of bytes which have to be copied
Return code	
–	-
Functional Description	
This function copies data from the CAN data buffer to the RAM.	
Particularities and Limitations	
This function can only be used within precopy functions.	

8.5.1.37 CanCopyToCan

CanCopyToCan

Prototype	
Single Receive Channel	`void CanCopyToCan ( CanChipDataPtr dst, void *src, vuint8 len)`
Multiple Receive Channel	
Parameter	
`dst`	Pointer to the destination CAN buffer or temporary buffer. This pointer is available in the Pretransmit Function.
`src`	Pointer to the source in default memory.
`len`	number of bytes which have to be copied
Return code	
–	-
Functional Description	
This function copies data from the RAM into the CAN data buffer.	
Particularities and Limitations	
This function can only be used within pretransmit functions.	

8.5.1.38 CanTxGetActHandle

CanTxGetActHandle

Prototype	
Single Receive Channel	`CanTransmitHandle CanTxGetActHandle (CanObjectHandle logTxHwObject)`
Multiple Receive Channel	
Parameter	

`logTxHwObject`	Handle of the CAN hardware transmit object. For indexed drivers this is a unique number over all CAN channels.
Return code	
`txObject`	Handle of the transmit object which is currently in the hardware transmit object. In case of enabled LowLevelMessageTransmit, this could also be a handle of such a message `kCanBufferMsgTransmit`: `CanCancelMsgTransmit` `kCanTxHandleNotUsed`: Handle is not valid
Functional Description	
This service functions returns the handle of the transmit message, which has been transmitted in a certain CAN hardware transmit object. The return value can be used as a parameter for `CanCancelTransmit()`. If the return value is `kCanBufferMsgTransmit`, `CanCancelMsgTransmit()` has to be called in stead of `CanCancelTransmit()`. `CanCancelTransmit()` ignores invalid handle and `kCanBufferMsgTransmit`.	
Particularities and Limitations	
This function is only allowed to be called in or after `ApplCanTxObjStart()` and before `ApplCanTxObjConfirmed()` of a certain CAN buffer.	

8.5.1.39 CanResetMsgReceivedCondition

CanResetMsgReceivedCondition

Prototype	
Single Receive Channel	`void` **`CanResetMsgReceivedCondition`** `( void )`
Multiple Receive Channel	`void` **`CanResetMsgReceivedCondition`** `( CanChannelHandle channel )`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
The service function `CanResetMsgReceivedConditional()` disables the calling of `ApplCanMsgCondReceived()`. For indexed CAN Driver, this functionality is related to the specified CAN channel.	
Particularities and Limitations	

8.5.1.40 CanSetMsgReceivedCondition

CanSetMsgReceivedCondition

Prototype	
Single Receive Channel	`void` **`CanSetMsgReceivedCondition`** `( void )`
Multiple Receive Channel	`void` **`CanSetMsgReceivedCondition`** `( CanChannelHandle channel )`
Parameter	

channel	Handle of a CAN channel. The generated macros should be used: kCanIndexX (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
The service function CanSetMsgReceivedConditional() enables the calling of ApplCanMsgCondReceived(). For indexed CAN Driver, this functionality is related to the specified CAN channel.	
Particularities and Limitations	

8.5.1.41 CanGetMsgReceivedCondition

CanGetMsgReceivedCondition

Prototype	
Single Receive Channel	void **CanGetMsgReceivedCondition** (void)
Multiple Receive Channel	void **CanGetMsgReceivedCondition** (CanChannelHandle channel)
Parameter	
channel	Handle of a CAN channel. The generated macros should be used: kCanIndexX (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
The service function CanGetMsgReceivedConditional()returns the status of the condition for calling ApplCanMsgCondReceived(). For indexed CAN Driver, this functionality is related to the specified CAN channel.	
Particularities and Limitations	

8.5.2 User Specific Functions

The user specific functions listed in this section are called by the CAN Driver and provided by the Application when certain events occur. The user can define user specific functions specifically for each message. The names in this section are only placeholders. The name could be set in the generation tool. The type of the particular user specific message must agree with the function types listed here.

8.5.2.1 UserPrecopy

UserPrecopy

Prototype	
Single Receive Channel	`vuint8` **`UserPrecopy`** `( CanRxInfoStructPtr rxStruct )`
Multiple Receive Channel	
Parameter	
`rxStruct`	Pointer to the receive structure
Return code	
`kCanCopyData`	Received data will be copied using the CAN Driver 's internal copy mechanism
`kCanNoCopyData`	CAN Driver doesn’t copy data and doesn’t perform indication
Functional Description	
User specific function of the CAN Driver, which is called in the receive interrupt of a CAN message before copying the data from the CAN Controller receive register to the application specific global data buffer. Depending on the function's return code, the CAN Driver will either terminate the processing of the received message (`kCanNoCopyData`) or resume normal processing (`kCanCopyData`).	
Particularities and Limitations	
■ For each CAN message a separate precopy function may be defined.	

8.5.2.2 UserIndication

UserIndication

Prototype	
Single Receive Channel	`void` **`UserIndication`**`( CanReceiveHandle rxObject)`
Multiple Receive Channel	
Parameter	
`rxObject`	Handle of the received message
Return code	
–	-
Functional Description	
User specific function which is called in the receive interrupt of a CAN message after data has been copied and the CAN Controller receive register have been released.	
Particularities and Limitations	
■ For each CAN message a separate indication function may be defined.	

8.5.2.3 UserPreTransmit

UserPreTransmit

Prototype	
Single Receive Channel	`vuint8 **UserPreTransmit**( CanTxInfoStruct txStruct )`
Multiple Receive Channel	
Parameter	
`txStruct`	Transmit structure
Return code	
`kCanCopyData`	After the return of this user specific function, the CAN Driver copies the data to be transmitted from the application specific global data buffer associated to the corresponding message to the CAN Controller transmit register
`kCanNoCopyData`	The CAN Driver does not copy data but starts the transmit request in the CAN Controller immediately
Functional Description	
User specific function which is called before the message is copied from the application specific data buffer to the transmit register of the CAN Controller. This is done in the scope of the Tx interrupt or the `CanTxTask()` via `CanTransmit(..)`. The usage of the internal copy mechanism of the CAN Driver is controlled by the return code. A possible usage is the acquiring and copying of existing data which are spread in the Application.	
Particularities and Limitations	
■ For each CAN message a separate pretransmit function may be defined.	

8.5.2.4 UserConfirmation

UserConfirmation

Prototype	
Single Receive Channel	`void **UserConfirmation**( CanTransmitHandle txObject )`
Multiple Receive Channel	
Parameter	
`txObject`	Handle of the transmit object
Return code	
–	-
Functional Description	
User specific function which is called in the scope of the CAN transmit interrupt routine or the `CanTxTask()` after the message has been sent on the CAN bus successfully	
Particularities and Limitations	
■ For each CAN message a separate confirmation function may be defined.	

8.5.3 Callback Functions

Callback functions are called by the CAN Driver on certain events and have to be provided by the Application. In contrast to the user specific functions in the section before the callback functions are not message related but only event related. Their name can also be reconfigured.

8.5.3.1 ApplCanBusOff

ApplCanBusOff

Prototype	
Single Receive Channel	void **ApplCanBusOff** (void)
Multiple Receive Channel	void **ApplCanBusOff** (CanChannelHandle channel)
Parameter	
channel	Handle of a CAN channel. The generated macros should be used: kCanIndexX (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
This callback function is called if the CAN Controller enters BusOff state. The function is called in the error interrupt, CanTask() or CanErrorTask().	
Particularities and Limitations	
If no Network Management is used which provides a BusOff error handling, the Application has to do the subsequent error handling (usually the re-initialization of the CAN Controller) by the CAN Driver service function CanResetBusOffStart() and CanResetBusOffEnd() itself.	

8.5.3.2 ApplCanWakeUp

ApplCanWakeUp

Prototype	
Single Receive Channel	void **ApplCanWakeUp** (void)
Multiple Receive Channel	void **ApplCanWakeUp** (CanChannelHandle channel)
Parameter	
channel	Handle of a CAN channel. The generated macros should be used: kCanIndexX (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
This callback function is called if a wake-up condition on the CAN bus is detected during sleep mode of the CAN Controller. The function is called in the wakeup interrupt, in the CanTask() or in the CanWakeupTask().	
Particularities and Limitations	

- If the CAN Controller was put into sleep mode by calling the service function `CanSleep()`, and afterwards there is a dominant level at the receive input of the CAN Controller, CAN Controller generates a wake-up interrupt. The CAN Driver calls the callback function `ApplCanWakeUp()` to handle further wake-up call activities, e.g. starting the Network Management.
- The Application must assure that the CAN transmit path is restored to its normal operating state, typically by the activation of the bus transceiver.
- This wake-up functionality is not supported by all CAN Controllers. If there is no power-down mode of the CAN Controller or if the microprocessor cannot detect an external wake-up condition by the CAN bus, this callback function will never be called.

8.5.3.3 ApplCanOverrun

ApplCanOverrun

Prototype	
Single Receive Channel	`void` **`ApplCanOverrun`** `( void )`
Multiple Receive Channel	`void` **`ApplCanOverrun`** `( CanChannelHandle channel )`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
This callback function is called if an overrun in a Basic CAN receive object was detected. It indicates a possible loss of receive data. The function is called in the error interrupt, in the receive interrupt, in the `CanTask()`, in the `CanRxTask()`, or in the `CanErrorTask()`.	
Particularities and Limitations	
The overrun is completely handled by the CAN Driver. This callback function only notifies the Application about such a condition.	

8.5.3.4 ApplCanFullCanOverrun

ApplCanFullCanOverrun

Prototype	
Single Receive Channel	`void` **`ApplCanFullCanOverrun`** `( void )`
Multiple Receive Channel	`void` **`ApplCanFullCanOverrun`** `( CanChannelHandle channel )`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
This callback function is called if an overrun of a Full CAN receive object was detected. It indicates a possible loss of receive data. The function is called in the error interrupt, in the receive interrupt, in the `CanTask()`, in the `CanRxTask()` or in the `CanErrorTask()`.	

Particularities and Limitations
■ The overrun is completely handled by the CAN Driver. This callback function only notifies the Application about an overrun.

8.5.3.5 ApplCanMsgReceived

ApplCanMsgReceived

Prototype	
Single Receive Channel	`vuint8 ApplCanMsgReceived( CanRxInfoStructPtr rxStruct )`
Multiple Receive Channel	
Parameter	
`rxStruct`	Pointer to receive information structure
Return code	
`kCanCopyData`	Receive processing will be continued
`kCanNoCopyData`	Receive processing will be terminated
Functional Description	
This callback function is called on every reception of a CAN message when the hardware acceptance filter is passed. The function is called in the receive interrupt, in the `CanTask()` or in the `CanRxTask()`.	
Particularities and Limitations	
■ The callback function may be used for gateway functionality or any other purpose. ■ There are preprocessor macros available to read the CAN identifier, the Data Length Code and the data in the CAN Controller receive register.	

8.5.3.6 ApplCanRangePrecopy

ApplCanRangePrecopy

Prototype	
Single Receive Channel	`vuint8 ApplCanRangePrecopy( CanRxInfoStructPtr rxStruct)`
Multiple Receive Channel	
Parameter	
`rxStruct`	Pointer to the receive structure
Return code	
`kCanCopyData`	The CAN receive interrupt routine is continued with verifying a match to the next range and ID search.
`kCanNoCopyData`	The CAN receive interrupt routine is terminated immediately after the CAN Controller is serviced
Functional Description	

This precopy function is not called on a specific message CAN identifier but on a complete CAN identifier range. The function is called in the receive interrupt, in the `CanTask()`, in the `CanRxTask()` or in `CanHandleRxMsg()`. The return code is not taken into account, if the range is handled via the RX Queue. In this case, the handling of the received message will be terminated after calling the range specific precopy function. The name of this function is only a placeholder. The name could be set in the generation tool. Up to four ranges with individual precopy functions can be specified per CAN channel.
Particularities and Limitations
■ Ranges are normally used for Network Management or Transport Protocol services only ■ In case a range configured to be handled via the Rx Queue, the return code of this function is ignored.

8.5.3.7 ApplCanAddCanInterruptDisable

ApplCanAddCanInterruptDisable

Prototype	
Single Receive Channel	`void` **`ApplCanAddCanInterruptDisable`** `( CanChannelHandle channel )`
Multiple Receive Channel	
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index) In case of Single Receive Channel `channel` is always 0.
Return code	
–	-
Functional Description	
Disabling of additional CAN interrupts (like separately implemented Wake-Up interrupts and Polling Tasks) can be added to the standard mechanism of the CAN by this callback function. The function is called on interrupt and task level.	
Particularities and Limitations	
■ `ApplCanAddCanInterruptDisable()` is only called if configured	

8.5.3.8 ApplCanAddCanInterruptRestore

ApplCanAddCanInterruptRestore

Prototype	
Single Receive Channel	`void` **`ApplCanAddCanInterruptRestore`** `( CanChannelHandle channel )`
Multiple Receive Channel	
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index) In case of Single Receive Channel `channel` is always 0.
Return code	
–	-
Functional Description	

Complementary callback function for `ApplCanAddCanInterruptDisable()`.The function is called on interrupt and task level.	
Particularities and Limitations	
■ `ApplCanAddCanInterruptRestore()` is only called if configured.	

8.5.3.9 ApplCanFatalError

ApplCanFatalError

Prototype	
Single Receive Channel	`void` **`ApplCanFatalError`** `( vuint8 errorNumber )`
Multiple Receive Channel	`void` **`ApplCanFatalError`** `( CanChannelHandle channel, vuint8 errorNumber )`
Parameter	
`errorNumber`	Error identification: There is a predefined list with supported assertion checks for each CAN Driver. All the function parameters starting with kError.... Please refer to chapter Assertions and the CAN Controller Specific documentation TechnicalReference_CAN_<hardware>.pdf [#hw_assert] for details.
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
If assertions are configured, the callback function `ApplCanFatalError(..)` is called in case of invalid user conditions (Application interface, reentrance), inconsistent generated data, hardware errors or internal errors (queue). An error number is passed by the parameter. The function is called on interrupt and task level.	
Particularities and Limitations	
■ This callback function does not have to return to the CAN Driver.	

8.5.3.10 ApplCanMsgNotMatched

ApplCanMsgNotMatched

Prototype	
Single Receive Channel Multiple Receive Channel	`void` **`ApplCanMsgNotMatched`** `( CanRxInfoStructPtr rxStruct )`
Parameter	
`rxStruct`	Pointer to the receive structure
Return code	
–	-
Functional Description	
This callback function is called if a CAN message passes the hardware acceptance filter, but not the software filter (inclusive the identifier specific predefined ranges). The function is called in the receive interrupt, in the `CanTask()` or in the `CanRxTask()`.	

Particularities and Limitations

8.5.3.11 ApplCanInit

ApplCanInit

Prototype	

Single Receive Channel	`void` **`ApplCanInit`**`( CanObjectHandle logTxHwObjectFirstUsed,`
	`CanObjectHandle logTxHwObjectFirstUnused)`

Multiple Receive Channel	`void` **`ApplCanInit`**`( CanChannelObject channel,`
	`CanObjectHandle logTxHwObjectFirstUsed,`
	`CanObjectHandle logTxHwObjectFirstUnused)`

Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
`logTxHwObjectFirstUsed`	Handle of the first CAN hardware transmit object of the current channel.

`logTxHwObjectFirstUnused`	Handle of the first unused CAN hardware transmit object of the current channel.
	example:
	`for ( i = logTxHwObjectFirstUsed; i < logTxHwObjectFirstUnused; i++)`
	`{`
	` /* loop over all used hardware transmit buffer of the current`
	`    channel */`
	`}`

Return code	
–	-
Functional Description	
This callback function is called in `CanInit()` for general purposes. In `CanInit()` transmit requests in the CAN Controller are canceled. This means the corresponding confirmation notification will never occur. User defined actions started in `ApplCanTxObjStart()` have to be stopped in `ApplCanInit()`.The function is called on interrupt and task level.	
Particularities and Limitations	
This callback is active only if ‘Tx observe’ functionality is activated.	

8.5.3.12 ApplCanTxObjStart

ApplCanTxObjStart

Prototype	
Single Receive Channel	void **ApplCanTxObjStart**(CanObjectHandle logTxHwObject)
Multiple Receive Channel	void **ApplCanTxObjStart**(CanChannelHandle channel, CanObjectHandle logTxHwObject)
Parameter	
channel	Handle of a CAN channel. The generated macros should be used: kCanIndexX (with X = 0 ... Number of generated channel index)
logTxHwObject	Handle of the CAN buffer transmit object. For indexed drivers this is a unique number over all CAN channels.
Return code	
–	-
Functional Description	
This callback function is called every time, a transmit request is initiated in the CAN Controller. This is done in the service CanTransmit(..).The function is called in the transmit interrupt, in the CanTask() or in the CanTxTask(), if the transmit queue is enabled.	
Particularities and Limitations	
This callback is active only if 'Tx observe' functionality is activated.	

8.5.3.13 ApplCanTxObjConfirmed

ApplCanTxObjConfirmed

Prototype	
Single Receive Channel	void **ApplCanTxObjConfirmed**(CanObjectHandle logTxHwObject)
Multiple Receive Channel	void **ApplCanTxObjConfirmed** (CanChannelHandle channel, CanObjectHandle logTxHwObject)
Parameter	
channel	Handle of a CAN channel. The generated macros should be used: kCanIndexX (with X = 0 ... Number of generated channel index)
logTxHwObject	Handle of the CAN buffer transmit object. For indexed drivers this is a unique number over all CAN channels.
Return code	
–	-
Functional Description	
This callback function is called every time, a successful transmission is confirmed by the CAN Controller in the scope of a transmit interrupt, in the CanTask() or in the CanTxTask().	
Particularities and Limitations	
This callback is active only if 'Tx observe' functionality is activated.	

8.5.3.16 ApplCanTimerEnd

ApplCanTimerEnd

Prototype	
Single Receive Channel	`void` **`ApplCanTimerEnd`**`(vuint8 timerIdentification)`
Multiple Receive Channel	`void` **`ApplCanTimerEnd`**`(CanChannelObject channel, vuint8 timerIdentification)`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
`timerIdentification`	Identifier for the hardware dependent loop timer
Return code	
–	-
Functional Description	
This callback function is called after a hardware dependent loop is finished, due to return value also of ApplCanTimerLoop or hardware condition met.	
Particularities and Limitations	

8.5.3.17 ApplCanGenericPrecopy

ApplCanGenericPrecopy

Prototype	
Single Receive Channel Multiple Receive Channel	`vuint8` **`ApplCanGenericPrecopy`**`(CanRxInfoStructPtr rxStruct)`
Parameter	
`rxStruct`	Pointer to the receive structure
Return code	
`kCanCopyData`	The UserPrecopy function will be called and the Received data will be copied using the CAN Driver's internal copy mechanism.
`kCanNoCopyData`	CAN Driver doesn't copy data and doesn't perform indication
Functional Description	
This precopy function is common to all receive messages. It will be called immediately after the DLC-check. The call of the `UserPrecopy` functions or copy of data are influenced by `ApplCanGenericPrecopy()`. The function is called in the receive interrupt, in the `CanTask()` or in the `CanRxTask()`.	
Particularities and Limitations	

8.5.3.18 ApplCanPreWakeup

ApplCanPreWakeup

Prototype

Single Receive Channel	`void` **`ApplCanPreWakeUp`**`( void )`
Multiple Receive Channel	`void` **`ApplCanPreWakeUp`**`(CanChannelHandle channel)`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
Is called just after the activation of the wakeup interrupt.	
Particularities and Limitations	

8.5.3.19 ApplCanTxConfirmation

ApplCanTxConfirmation

Prototype	
Single Receive Channel	`void` **`ApplCanTxConfirmation`**`( CanTxInfoStructPtr txStruct)`
Multiple Receive Channel	
Parameter	

`txStruct`	Pointer to transmit structure
	
	`typedef struct`
	`{`
	`  CanChannelHandle   Channel;`
	`  CanTransmitHandle  Handle;`
	`} tCanTxConfInfoStruct;`
	
	`typedef tCanTxConfInfoStruct`
	`*CanTxInfoStructPtr;`
	
	`Handle:`
	- 0 ... (kCanNumberOfTxMessages-1): the handle of the Tx message.
	- `kCanBufferMsgTransmit`, in case the message was sent via CanCancelMsgTransmit().
	- `((CanTransmitHandle)0xFFFFFFFEU)`
	in case the message was cancel by CanCanTransmit() or CanCancelMsgTransmit() but sent on the bus

Return code	
–	-
Functional Description	
This confirmation function is common to all transmit messages. It will be called after the successful transmission. The function is called in the transmit interrupt, in the `CanTask()` or in the `CanTxTask()`.	
Particularities and Limitations	

8.5.3.20 ApplCanMsgDlcFailed

ApplCanMsgDlcFailed

Prototype	
Single Receive Channel	`void` **`ApplCanMsgDlcFailed`**`( CanRxInfoStructPtr rxStruct )`
Multiple Receive Channel	
Parameter	
`rxStruct`	Pointer to the receive structure
Return code	
–	-
Functional Description	
This callback function is called, if the DLC check fails. To activate this callback function the switch `C_ENABLE_DLC_FAILED_FCT` has to be set in a user configuration file. The function is called in the receive interrupt, in the `CanTask()` or in the `CanRxTask()`.	
Particularities and Limitations	
`It depends on the OEM if ApplCanMsgDlcFailed()` is available.	

8.5.3.21 ApplCanCancelNotification

ApplCanCancelNotification

Prototype	
Single Receive Channel	`void` **`ApplCanCancelNotification`**`(CanTransmitHandle txHandle)`
Multiple Receive Channel	
Parameter	
`txHandle`	Handle of cancelled transmit object
Return code	
–	-
Functional Description	
This function will be called if a transmit message is deleted (CanCancelTransmit, CanOffline or CanInit). This function could be called in Interrupt or Task context.	
Particularities and Limitations	
`ApplCanCancelNotification()` is only called if configured.	

8.5.3.22 ApplCanOnline

ApplCanOnline

Prototype	
Single Receive Channel Multiple Receive Channel	`void ApplCanOnline(CanChannelHandle channel)`
Parameter	
`channel`	CAN Channel on which the CAN driver was switched to online mode.
Return code	
–	-
Functional Description	
This callback function indicates that the CAN driver is switched to online mode. This function is called by the CAN Driver if the mode change is initiated via `CanOnline()`.	
Particularities and Limitations	
Call context: This function is called within `CanOnline()`. This service function is only allowed to be called on task level.	

8.5.3.23 ApplCanOffline

ApplCanOffline

Prototype	
Single Receive Channel Multiple Receive Channel	`void ApplCanOffline(CanChannelHandle channel)`
Parameter	
`channel`	CAN Channel on which the CAN driver was switched to offline mode.
Return code	
–	-
Functional Description	
This callback function indicates that the CAN driver is switched to offline mode. This function is called by the CAN Driver if the mode change is initiated via `CanOffline()`.	
Particularities and Limitations	
Call context: This function is called within `CanOffline()`. This service function is allowed to be called on task level or on interrupt level.	

8.5.3.24 ApplCanMsgCondReceived

ApplCanMsgCondReceived

Prototype	
Single Receive Channel Multiple Receive Channel	`void ApplCanMsgCondReceived (CanRxInfoStructPtr rxStruct)`
Parameter	
`rxStruct`	Pointer to the receive information structure

Return code	
–	-
Functional Description	
This callback function is conditionally called on every reception of a CAN message when the hardware acceptance filter is passed. There are preprocessor macros available to read the CAN identifier, the Data Length Code and the data in the CAN Controller receive register.	
Particularities and Limitations	
Call context: The function is called in the receive interrupt, in the `CanTask()` or in the `CanRxTask().`	

8.5.3.25 ApplCanMemCheckFailed

ApplCanMemCheckFailed

Prototype	
Single Receive Channel	`vuint8` **`ApplCanMemCheckFailed`** `( void )`
Multiple Receive Channel	`vuint8` **`ApplCanMemCheckFailed`** `( CanChannelHandle channel )`
Parameter	
`Channel`	Handle of the CAN channel on which the check failed. The generated macros should be used: **`kCanIndexX`** (with X = 0 ... Number of generated channel index)
Return code	
`kCanEnableCommunication`	Allow communication.
`kCanDisableCommunication`	Disable communication, no reception and no transmission is performed.
Functional Description	
This callback function is called if the CAN driver has found at least one corrupt memory bit within the CAN mailboxes. The application can decide if the CAN driver allows further communication by means of the return value.	
Particularities and Limitations	
Call context: This function is called on task level or within the busoff interrupt.	

8.5.3.26 ApplCanCorruptMailbox

ApplCanCorruptMailbox

Prototype	
Single Receive Channel	`void` **`ApplCanCorruptMailbox`** `( CanObjectHandle hwObjHandle )`

Multiple Receive Channel	`void` **`ApplCanCorruptMailbox`** `( CanChannelHandle channel,`
	`CanObjectHandle hwObjHandle )`

Parameter	
`hwObjHandle`	The index of the corrupt mailbox.
`channel`	Handle of the CAN channel on which the corrupt mailbox is located. The generated macros should be used: **`kCanIndexX`** (with X = 0 ... Number of generated channel index)

Return code	
–	-
Functional Description	
This callback function is called if the CAN driver has found a corrupt mailbox.	
Particularities and Limitations	
Call context: This function is called on task level or within the busoff interrupt.	

9 Description of the API (High End extension)

9.1 Functions

9.1.1 Service Functions

9.1.1.1 CanTxObjTask

CanTxObjTask

Prototype	
Single Receive Channel	`void CanTxObjTask ( CanObjectHandle txObjHandle )`
Multiple Receive Channel	`void CanTxObjTask ( CanChannelHandle canHwChannel, CanObjectHandle txObjHandle )`
Parameter	
`canHwChannel`	Handle of a CAN Hardware channel. The generated macros should be used: Normal Tx Object: `C_TX_NORMAL_<channel>_HW_CHANNEL` (with <channel> = 0 ... Number of logical channel) Full CAN Tx Object: `<message name>_HW_CHANNEL` (with <message name> = Name of the Message with pre and postfixes generated in “can_par.h”t) Low Level Tx Object: `C_TX_LL_<channel>_HW_CHANNEL` (with <channel> = 0 ... Number of logical channel)
`txObjHandle`	Handle of a Tx mailbox. The generated macros should be used: Normal Tx Object: `C_TX_NORMAL_<channel>_HW_OBJ` (with <channel> = 0 ... Number of logical channel) Full CAN Tx Object: `<message name>_HW_OBJ` (with <message name> = Name of the Message with pre and postfixes generated in “can_par.h”) Low Level Tx Object: `C_TX_LL_<channel>_HW_OBJ` (with <channel> = 0 ... Number of logical channel)
Return code	
–	-
Functional Description	
The service function `CanTxObjTask()` does polling of specified transmit hardware objects in the CAN controller. Confirmation functions will be called and confirmation flags will be set. If the transmit queue is configured, this service function additionally transmits the queued messages.	
Particularities and Limitations	
■ `CanTxObjTask()` is available, if the individual polling mode and at least one mailbox is configured for polling.	

9.1.1.2 CanRxFullCANObjTask

CanRxFullCANObjTask

Prototype	
Single Receive Channel	`void CanRxFullCANObjTask (CanObjectHandle rxObjHandle )`
Multiple Receive Channel	`void CanRxFullCANObjTask ( CanChannelHandle canHwChannel, CanObjectHandle rxObjHandle )`
Parameter	
canHwChannel	Handle of a CAN Hardware channel. The generated macros should be used: `<message name>_HW_ CHANNEL` (with <message name> = Name of the Message with pre and postfixes generated in “can_par.h”)
rxObjHandle	Handle of an Rx mailbox. The generated macros should be used: `<message name>_HW_OBJ` (with <message name> = Name of the Message with pre and postfixes generated in “can_par.h”t)
Return code	
–	-
Functional Description	
The service function `CanRxFullCANObjTask()` does polling of specified Full CAN receive objects according to the configured objects in polling mode.	
Particularities and Limitations	
■ `CanRxFullCANObjTask()` must not run on higher priority than other CAN functions. ■ `CanRxFullCANObjTask()` is available, if the individual polling mode and at least one mailbox is configured for polling.	

9.1.1.3 CanRxBasicCANObjTask

CanRxBasicCANObjTask

Prototype	
Single Receive Channel	`void CanRxBasicCANObjTask ( void )`
Multiple Receive Channel	`void CanRxBasicCANObjTask (CanChannelHandle canHwChannel, CanObjectHandle rxObjHandle )`
Parameter	
canHwChannel	Handle of a CAN Hardware channel. The generated macros should be used: `C_BASIC<number_of_the_BasicCAN>_<channel>HW_CHANNEL` (with <number_of_the_BasicCAN>= the logical number of the Basic CAN on this channel <channel> = 0 ... Number of logical channel)
txObjHandle	Handle of a Rx mailbox. The generated macros should be used: `C_BASIC<number_of_the_BasicCAN>_<channel>_HW_OBJ` (with <number_of_the_BasicCAN>= the logical number of the Basic CAN on this channel <channel> = 0 ... Number of logical channel)
Return code	
–	-
Functional Description	

The service function `CanRxBasicCANObjTask()` does polling of specified Basic CAN receive objects according to the configured objects in polling mode.
Particularities and Limitations
■ `CanRxBasicCANObjTask()` must not run on higher priority than other CAN functions. ■ `CanRxBasicCANObjTask()` is available, if the individual polling mode and at least one mailbox is configured for polling.

9.1.1.4 CanMsgTransmit

CanMsgTransmit

Prototype	
Single Receive Channel	`vuint8 CanMsgTransmit ( tCanMsgTransmitStruct *txData )`
Multiple Receive Channel	`vuint8 CanMsgTransmit ( CanChannelHandle channel, tCanMsgObject *txData )`
Parameter	
`*txData`	Pointer to the structure with ID, DLC and data to send
`channel`	Handle of a CAN channel. The generated macros should be used: **`kCanIndexX`** (with X = 0 ... Number of generated channel index)
Return code	
`kCanTxOk`	if the message-buffer is free and the data could be copied to the CAN-data-buffer or if passive mode (`CanSetPassive()`) is active.
`kCanTxFailed`	if offline mode is active, the CAN buffer is not free.
Functional Description	
This function is called by the application. The function sends the message which is defined in the txData (Id, DLC, Data) to the CAN-Bus which is defined in the channel.	
Particularities and Limitations	
■ the contents of txData may not be changed while `CanMsgTransmit()` is running	

9.1.1.5 CanCancelMsgTransmit

CanCancelMsgTransmit

Prototype	
Single Receive Channel	`void CanCancelMsgTransmit( void )`
Multiple Receive Channel	`void CanCancelMsgTransmit( CanChannelHandle channel )`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
The call of `ApplCanMsgTransmitConf()` is suppressed, if a message is already in the transmit buffer of the CAN controller associated with `CanMsgTransmit()`. Dependent on the configuration this function cancels a message in the CAN hardware.	

Particularities and Limitations
The function call of `CanCancelTransmit()` must not interrupt the transmit ISR, `CanMsgTransmit()` or the `CanTxTask()`. Though a transmission is canceled it will be sent if the request has been already in the hardware object. Only if activated and highly dependent on hardware and vehicle manufacturer the transmit request which is initiated with `CanMsgTransmit()` can be deleted in the hardware transmit object, too. The function `CanCancelMsgTranmit()` is only available if the confirmation `ApplCanMsgTransmitConf()` is configured.

9.1.1.6 CanHandleRxMsg

CanHandleRxMsg

Prototype	
Single Receive Channel	`void CanHandleRxMsg ( void )`
Multiple Receive Channel	
Parameter	
–	-
Return code	
–	-
Functional Description	
The service function `CanHandleRxMsg()` handles the received messages which are stored in the Rx Queue. The standard mechanism (GenericPrecopy, UserPrecopy, copy of data, Indication Flag and UserIndication) is started for each stored message.	
Particularities and Limitations	
This function is only allowed to be called on task level.	

9.1.1.7 CanDeleteRxQueue

CanDeleteRxQueue

Prototype	
Single Receive Channel	`void CanDeleteRxQueue ( void )`
Multiple Receive Channel	
Parameter	
–	-
Return code	
–	-
Functional Description	
The service function `CanDeleteRxQueue()` clears all pending messages in the Rx Queue.	
Particularities and Limitations	
This function is only allowed to be called on task level.	

9.1.2 Callback Functions

9.1.2.1 ApplCanMsgTransmitConf

ApplCanMsgTransmitConf

Prototype	
Single Receive Channel	`void` **`ApplCanMsgTransmitConf`** `( void )`
Multiple Receive Channel	`void` **`ApplCanMsgTransmitConf`** `(CanChannelHandle channel)`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
This function will be called from the CAN Driver after sending the message. It is called directly in the transmit Interrupt (Confirmation) from the CAN-Controller. On task level it is called in the `CanTask()` or in the `CanTxTask()`. With this callback function it is possible to implement a queue-functionality.	
Particularities and Limitations	
`ApplCanMsgTransmitConf()` is only called if configured.	

9.1.2.2 ApplCanMsgTransmitInit

ApplCanMsgTransmitInit

Prototype	
Single Receive Channel	`void` **`ApplCanMsgTransmitInit`** `( void )`
Multiple Receive Channel	`void` **`ApplCanMsgTransmitInit`** `(CanChannelHandle channel)`
Parameter	
`channel`	Handle of a CAN channel. The generated macros should be used: `kCanIndexX` (with X = 0 ... Number of generated channel index)
Return code	
–	-
Functional Description	
This function will be called from the CAN Driver after a possible cancel of a transmit request in `CanInit()`.	
Particularities and Limitations	
`ApplCanMsgTransmitInit()` is only called if `ApplCanMsgTransmitConf()` is configured.	

9.1.2.3 ApplCanMsgCancelNotification

ApplCanMsgCancelNotification

Prototype	
Single Receive Channel	`void` **`ApplCanMsgCancelNotification`** `( void )`

Multiple Receive Channel	`void ApplCanMsgCancelNotification( CanChannelHandle channel )`
Parameter	
`Channel`	CAN Channel on which the Tx Object was cancelled.
Return code	
–	-
Functional Description	
This function will be called if a transmit message is deleted (CanCancelMsgTransmit). It applies only to Tx messages that have been transmitted via CanMsgTransmit. This function could be called in Interrupt or Task context.	
Particularities and Limitations	
`ApplCanMsgCancelNotification()` is only called if configured.	

9.1.2.4 ApplCanPreRxQueue

ApplCanPreRxQueue

Prototype	
Single Receive Channel	`vuint8 ApplCanPreRxQueue( CanRxInfoStructPtr rxStruct )`
Multiple Receive Channel	
Parameter	
`rxStruct`	Pointer to receive information structure
Return code	
`kCanCopyData`	The data of the received message will be stored in the Rx Queue.
`kCanNoCopyData`	The data of the received message are not stored in the Rx Queue. The reception is handled within the receive interrupt.
Functional Description	
This precopy function is called if a message is received which is a valid message in the receive structures or has to be handled via a range (in case the use of the Rx Queue is configured for this range). The application can decide whether to handle the message via the Rx Queue or the standard CAN Driver mechanism within the receive interrupt.	
Particularities and Limitations	
Call context: This function is called within the receive interrupt.	

9.1.2.5 ApplCanRxQueueOverrun

ApplCanRxQueueOverrun

Prototype	
Single Receive Channel	`void ApplCanRxQueueOverrun( void )`
Multiple Receive Channel	
Parameter	
–	-
Return code	
–	-

Functional Description
This callback function indicates an overrun of the Rx Queue. This function is called by the CAN Driver, in case a new message has to be stored in the Rx Queue, but the Queue if full. This new message will be lost.
Particularities and Limitations
Call context: This function is called within the receive interrupt.

10 Configuration (Standard and High End)

This chapter describes the common options for configuring (customizing) the CAN Driver. CAN Controller dependent configuration is described in the CAN Controller specific documentation TechnicalReference_CAN_<hardware>.pdf [#hw_conf]. The configuration can be done by the Generation Tool automatically.

10.1 Network Database – Attribute Definition

⚠	**Caution** Attribute names in CANgen are case sensitive and not evaluated, if the name case is incorrect.

Name	GenMsgMinAcceptLength
Description	The DLC check can be configured to verify the received DLC against the value given by this attribute (Against minimum acceptance length). The value can be smaller than the Application receive buffer of this message. Value “-1” means the DLC of the received message will be compared to the length of the Application receive buffer of this message.
Type Of Object	Message
Value Type	Integer
Default	-1
Minimum	-1
Maximum	8

10.2 Automatic Configuration by GENy

Using the Generation Tool GENy the configuration can be done by the tool. The configuration options common to all CAN Drivers is described here. The CAN Controller dependent options are described in the CAN Controller specific user manual. The following dialog describes the CAN Driver common options. The configuration data is stored by the tool in the file can_cfg.h for GENy.

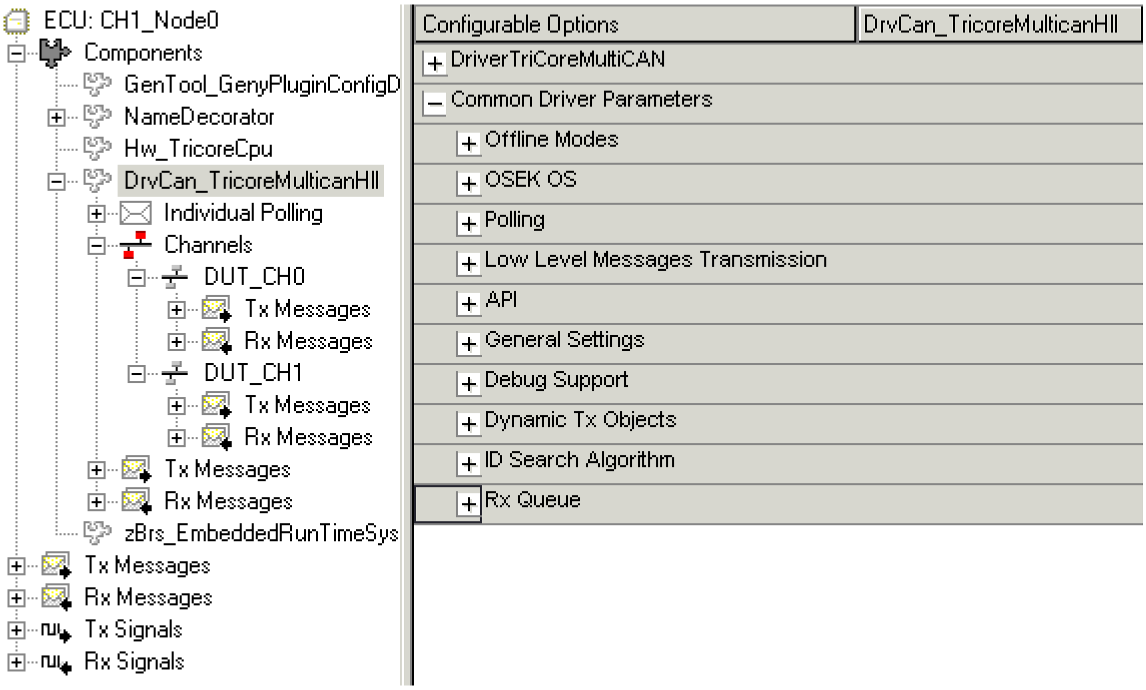


Figure 10-1 Configuration of the common CAN Driver options with GENy

Feature	Explanation
Common Driver Parameters	
Online/Offline Callback Functions	If `CanOnline()` or `CanOffline()` is called the Application is notified that a certain CAN driver state was entered.
DLC check	The DLC check can be configured to “disabled”, comparison of received DLC “against data length” or “against minimum acceptance length”. “against data length” means the number of bytes necessary for the ECU and is equal to the length of the application data buffer. The minimum acceptance length can be configured message specific via database or on the Rx message view. If DLC check is used, received Data messages with smaller DLC than expected for this ID are ignored.
Data Copy Mechanism	The Data Copy Mechanism can be configured to “copy number of needed bytes” or “copy all received bytes”. “copy number of needed bytes” means the CAN driver copies always the number of bytes which is equal to the length of the application data buffer. “copy all received bytes” means if the number of received bytes is less than the size of the application data buffer only the received bytes are copied. Otherwise the number of bytes which is equal to the length of the application data buffer will be copied.
RAM check	The CAN driver supports a RAM check of the CAN controller mailboxes.
Sleep / Wakeup	If this field is checked, the CAN controller can be switched to sleep mode.

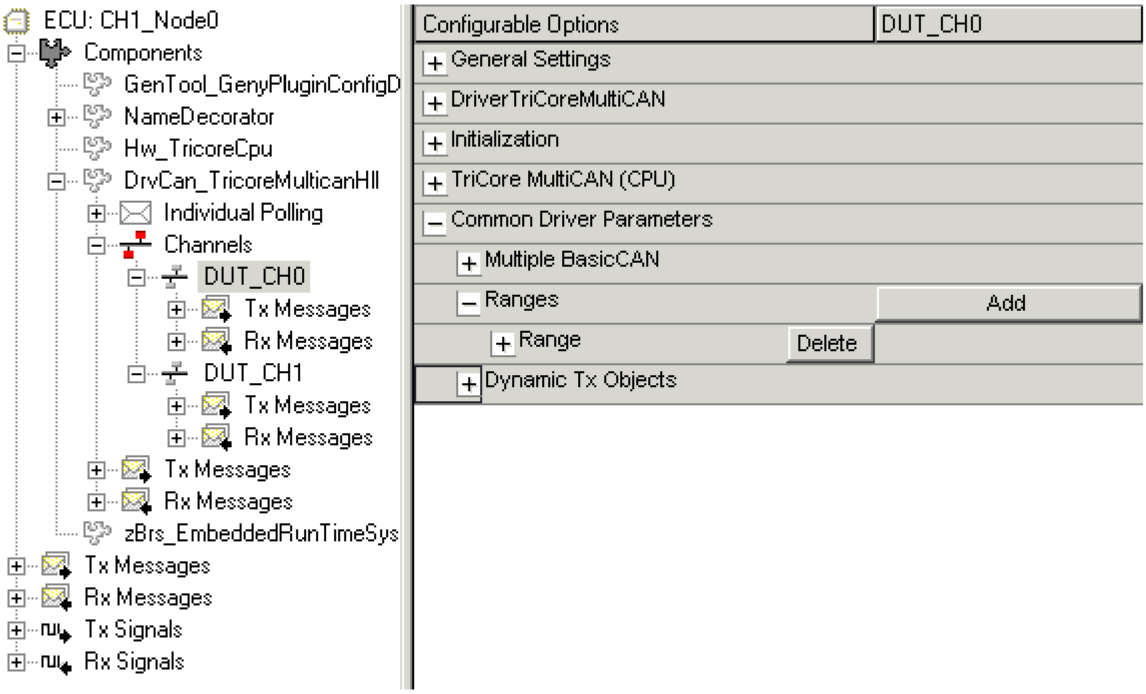
Feature	Explanation
Cancel in Hardware	`CanCancelTransmit()` and `CanCancelMsgTransmit()` will delete a pending request in the CAN controller hardware.
FullCAN Overrun Notification	If checked an overrun in the receive FullCAN objects will be signaled to the Application.
Receive Function	The receive function `ApplCanMsgReceived()` is called by the CAN Driver on every reception of a CAN message after the hardware acceptance filter is passed. Within this callback function the Application may preprocess the received message in any way (ECU specific dynamic filtering mechanisms, preprocessing of the messages, gateway functionality...). If the callback function `ApplCanMsgReceived()` returns `kCanCopyData`, the CAN Driver continues to work on the message received. If the callback function returns `kCanNoCopyData`, the CAN finishes working on the message received. The callback function has to be defined by the application.
Active / Passive State	If this field is checked, the CAN Driver's transmit path can be switched to the "Passive" state. In this state no CAN messages are sent to the CAN bus. The transmit request always returns with ”OK”. This ”passive” state functionality may be used to localize errors in a CAN bus: If there are errors in a CAN bus, and the errors disappear when a particular node is switched to passive state, the scapegoat is found. The Application must be switched to the passive state and back to active state by an external tester. If active/passive state is enabled, the CAN Driver is in active state after initialization. If this field is not selected, the corresponding service functions are available but without any effect on the CAN Driver status.
Extended Status	This is the global checkbox for using hardware status information in the CAN Driver service `CanGetStatus()` or not.
Transmit Queue	If this field is checked the CAN Driver is configured to use a transmit queue. If no transmit queue is used, the Application is responsible to restart a transmit request if it wasn’t accepted by the CAN Driver. In the case of using a transmit queue, a transmit request is almost accepted. But the queue does only store the transmit request of a message. It doesn’t store the data to be sent in any case. The CAN Driver inserts a transmit request to the queue, if no hardware object is available when `CanTransmit()` is called. On a transmit interrupt, this means a former requested message is transmitted, the CAN Driver checks whether transmit requests are stored in the queue. If so, these requests are removed from the queue and the transmit request is executed. The search algorithm in the queue is priority based, there is no FIFO strategy. This means the identifier with the lowest number is removed first from the queue.
Tx observation	This is the global switch for using the Tx observe functionality of the CAN Driver or not.

Feature	Explanation
Message-Not-Matched Function	This is the global switch for using the `MsgNotMatched` function of the CAN Driver or not.
Overrun Notification	If this field is checked the CAN Driver is configured to notify the Application in case of an overrun. The Application has to provide an Overrun callback function: void `ApplCanOverrun(void)` /* Overrun in the CAN Controller */ The overrun handling itself is done by the CAN Driver.
Hardware Loop Check	This is the global switch for using the hardware loop check of the CAN Driver or not.
Partial Offline Mode	If the checkbox “Use PartOffline Functionality” is checked, the partial offline mode is available.
Generic Pre-copy	This checkbox enables the use of the generic precopy function – `ApplCanGenericPrecopy()`. This precopy function is common to all receive messages. it will be called as soon as the receive handle is determined.
CAN Copy from & to CAN	This checkbox enables the use of copy functions `CopyFromCan()` and `CopyToCan()`.
CAN cancel Notification	The application will be notified via `ApplCanCancelNotification()` if a message is canceled and therefore confirmation will occur. This is valid for messages which have been requested via `CanTransmit()`.
CAN Interrupt Control Callbacks	There are two call back functions for the application. Within `CanCanInterruptDisable()` the function `ApplCanAddCanInterruptDisable()` is called and within `CanCanInterruptRestore()` the function `ApplCanAddCanInterruptRestore()` is called. These two functions have to be used to handle the wake-up interrupt if the hardware treats this interrupt separately or if the Driver runs in Polling Mode the polling tasks have to be disabled.
Common Confirmation Function	If this field is checked the common confirmation function of the CAN Driver is enabled.
Offline Modes	
Name of Mode X	with X = 0..7. If the partial Offline Mode is enabled the name of each send group can be configured here. more...
Default Mapping	
Message Class 0	All messages with don’t belong to an other class are assigned to this class.
OfflineModeX	with X = 0..7. The message class 0 can be assigned to certain Offline Modes (send goups) by selecting the check box.
Message Class 1 (Appl)	All application messages are assigned to this message class.
OfflineModeX	with X = 0..7. The message class 1 can be assigned to certain Offline Modes (send goups) by selecting the check box.

Feature	Explanation
Message Class 2 (NM)	All network mangement messages are assigned to this message class.
OfflineModeX	with X = 0..7. The message class 2 can be assigned to certain Offline Modes (send goups) by selecting the check box.
Message Class 3 (TP)	All transport layer messages are assigned to this message class.
OfflineModeX	with X = 0..7. The message class 3 can be assigned to certain Offline Modes (send goups) by selecting the check box.
Message Class 4 (Diag)	All diagnostic messages are assigned to this message class.
OfflineModeX	with X = 0..7. The message class 4 can be assigned to certain Offline Modes (send goups) by selecting the check box.
Message Class 5 (IL)	All interaction layer messages are assigned to this message class.
OfflineModeX	with X = 0..7. The message class 5 can be assigned to certain Offline Modes (send goups) by selecting the check box.
OSEK OS	
Use OsekOS Interrupt Cat 2	In case of using OSEK-OS the interrupt category of the CAN Driver interrupts have to be defined. Normally category 1 is used. Instead of this category 2 can be selected.
OSEK OS	If this field is checked the CAN Driver is configured to support OSEK-OS. The kind of OSEK-OS depends on the specific microprocessor.
Polling	
Polling type	The polling type can be switched between “None”, “Type specific” and “individual”. Individual: If enabled, each BasicCAN, Normal Tx, Low level Tx and FullCAN can be selected to be polled individual
Rx Basic CAN Polling	Normally the CAN driver works interrupt driven. To use the reception via Basic CAN objects in polling mode, check this field. This field is available in “Type specific polling mode”.
Rx Full CAN Polling	Normally the CAN driver works interrupt driven. To use the reception via Full CAN objects in polling mode, check this field. This field is available in “Type specific polling mode”.
Tx Polling	Normally the CAN driver works interrupt driven. To use the transmission in polling mode, check this field. This field is available in “Type specific polling mode”.
Wakeup Polling	Normally the CAN driver works interrupt driven. To use the wake up detection in polling mode, check this field.
CAN Status Polling	Normally the CAN driver works interrupt driven. To use the Status and Error detection in polling mode, check this field.
Low Level Transmission	

Feature	Explanation
Cancel Notification Function	The application will be notified if a message is canceled and therefore confirmation will occur. This is valid if the transmission has been requested via `CanMsgTransmit()`.
Enable Low Level Transmission	If the checkbox “Use Low Level Message Transmit” is checked, the function `CanMsgTransmit()` can be used.
Confirmation Function	This checkbox is only available, if “use Low Level Message Transmit” is active. The confirmation and init callback functions of low level transmit functionality can be activated by this checkbox.
API	
Symbolic Names for Signal Values	If the database allows the assignment of value tables to individual signals, this feature is selectable. If this functionality is enabled, symbolic names for values are generated for all signals that have an associated value table
Indexed Component	This switch determines whether the component should configure the indexed or non-indexed version of the driver component.
General Settings	
Security level	This is the define value to configure the security level. Valid values are 0, 10, 20 or 30. more...
User Config File	The CAN Driver configuration file (can_cfg.h) is generated. If the user wants to overwrite this automatically generated configuration file, the user is able to define the name of a user defined configuration file which is included at the end of the generated file can_cfg.h. This means entries in the user defined configuration file overwrite the entries in can_cfg.h.
Debug Suport	
Assertions	There are different groups of assertions supported by the CAN Driver. They can be selected depending of the development phase: **None**: No debug functionality active. **User**: User API is debugged. The CAN Driver service function parameters are checked. **Hardware**: The CAN Controller interface is checked. Depends on CAN Controller. **Gen**: The configuration data are checked. **Internal**: CAN Driver internal checking.
Dynamic Tx Objects	
ID	If the checkbox “ID” is checked, the IDs of the dynamic transmit objects can be changed by the service function `CanDynTxObjSetId()` and/or by the service function `CanDynTxObjSetExtId()`. The last mentioned service function is only available if extended ID addressing is checked in the Generation Tool.
DLC	If the checkbox “DLC” is checked, the DLC of the dynamic transmit objects can be changed by the service function `CanDynTxObjSetDlc`.
Data Pointer	If the checkbox “Data Pointer” is checked, the data pointers of the dynamic transmit objects can be changed by the service function `CanDynTxObjSetDataPtr()`.

Feature	Explanation
Confirmation	If the checkbox “Confirmation” is checked, the confirmation function of the dynamic transmit objects can be changed by the service function `CanDynTxObjSetConfirmationFct()`. The occurrence of this switch is CAN Controller dependent.
Pre-transmit	If the checkbox “Pre-transmit” is checked, the pretransmit function of the dynamic transmit objects can be changed by the service function `CanDynTxObjSetPreTransmitFct()`.The occurrence of this switch is CAN Controller dependent.
ID Search Algorithm	
Search Algorithm	For a Basic CAN Controller or the Basic CAN object of a Full CAN Controller the hardware acceptance filtering provided by the CAN Controller is not sufficient. Therefore a software acceptance filtering has to be supported by comparing the incoming message identifier with the complete list of all relevant message identifier. Here could the way how to search in the table of the receive messages be defined. The optimum algorithm depends on the number of the received messages to search in and their identifier structure. The supported search algorithms are dependent of the CAN Driver and the hardware. Refer to the CAN controller specific documenation TechnicalReference_CAN_<hardware>.pdf [#hw_feature] for more information. - linear - hash search - index search - table search
Additional Memory [Byes]	This shows the byte consumption when **hash search** is selected. It is just for information.
Maximum Search Steps	This is only activated when **hash search** is selected. Enter here the amount of maximum search steps that are necessary to find the received message ID in the list of to be received messages. The little this value the greater the additional memory bytes and the faster the receive ISR.
Rx Queue	
Overrun Notification	The Application is informed, if the Rx Queue is full and a new message should be copied into the Queue. The new message will be lost.
Enable Rx Queue	If the checkbox “Enable Rx Queue” is checked, the RX Queue is enabled. Else the Rx Queue is disabled.
Pre Rx Queue Function	If the checkbox “Pre Rx Queue Function” is checked, a Callback function is enabled where the application could decide what should happen with the CAN Message which was received. The two possibilities are to store the message in the queue, or to process the message in the interrupt.
Number of queued Rx Messages	Specifies the depth of the queue ("Number of queued Rx messages").



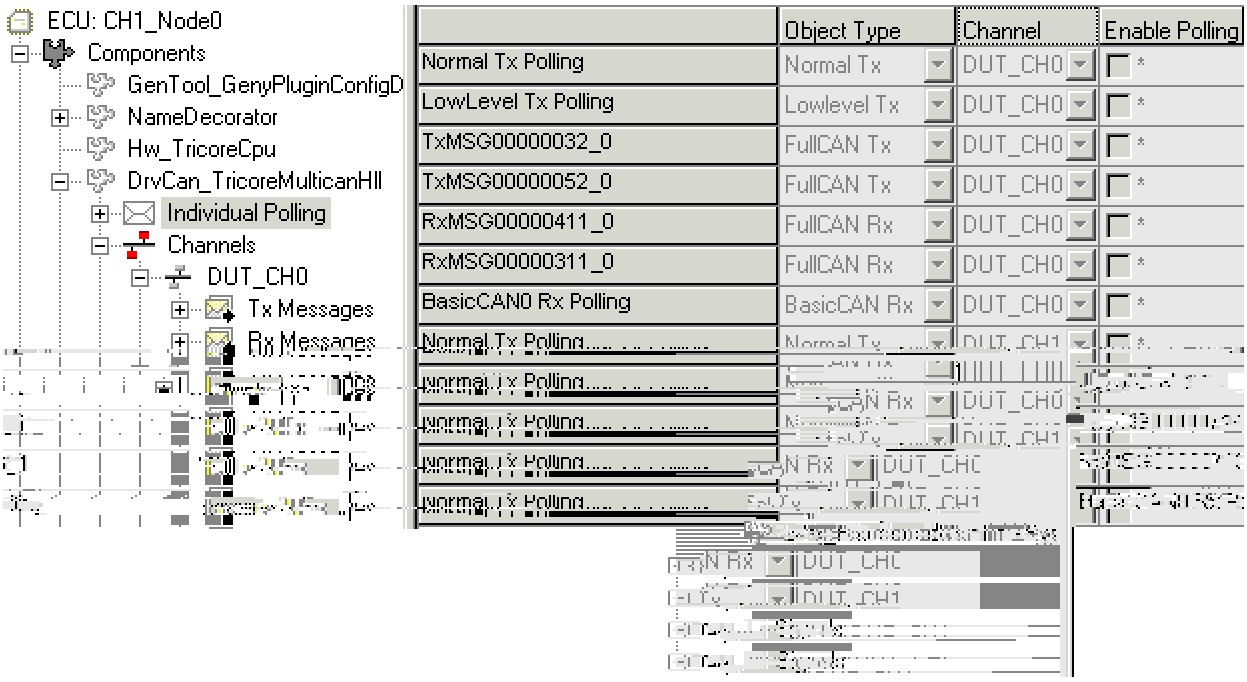
ECU: CH1_Node0
- Components
  - GenTool_GenyPluginConfigD
  - NameDecorator
  - Hw_TricoreCpu
  - DrvCan_TricoreMulticanHll
    - Individual Polling
    - Channels
      - DUT_CH0
        - Tx Messages
        - Rx Messages
      - DUT_CH1
        - Tx Messages
        - Rx Messages
    - Tx Messages
    - Rx Messages
  - zBrs_EmbeddedRunTimeSys
- Tx Messages
- Rx Messages
- Tx Signals
- Rx Signals

Configurable Options	DUT_CH0
+ General Settings	
+ DriverTriCoreMultiCAN	
+ Initialization	
+ TriCore MultiCAN (CPU)	
− Common Driver Parameters	
+ Multiple BasicCAN	
− Ranges	Add
+ Range Delete	
+ Dynamic Tx Objects	

Figure 10-2 Channel Specific Configuration for GENy

Features	Explanation
Configuration Options	
General Settings	
Bus System Type	Each channel is configured for a specific type of bus system. The bus system is always CAN for CAN Driver.
Manufacturer	Manufacturer with is specified in the database file for this channel.
Common Driver Parameters	
Multiple Basic CAN	
Enable Multiple Basic CAN	If checked, the number of Basic CAN objects can be selected. The deselecting of this checkbox resets the number of Basic CAN objects to the default.
Number Of BasicCAN Objects	Enter number of needed Basic CAN objects. Each Basic CAN object may consist of 2 hardware mailboxes. This depends on the CAN controller.
Ranges / Range Precopy Functions	
Range	Ranges are normally used for Network Management, Transport Protocol and so on. There are in maximum 4 ranges configurable.
Mask	Acceptance mask of this identifier range.
Code	Acceptance code of this identifier range.
Precopy function	Specific precopy function for this range.
Extended IDs	The range can be specified to receive extended IDs. If not selected, this range will receive standard IDs.

Use Own Filter	If there are enough Basic CAN filters available, one Basic CAN filter can be used exclusive for this range.
Use Rx Queue	All messages which are received by this range can be configured to be handled via the Rx Queue. This is only possible, if the feature Rx Queue is activated.
Dynamic Tx objects	
Number of dynamic Tx objects	Maximum number of dynamic send objects which are available at run time.

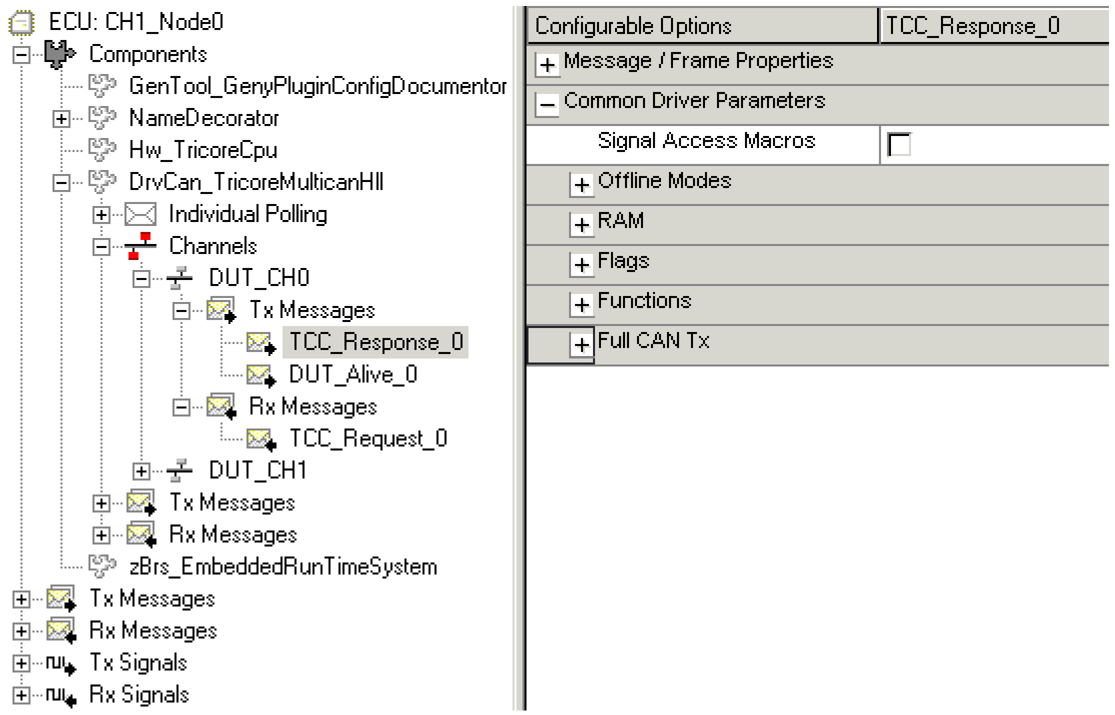


ECU: CH1_Node0
- Components
  - GenTool_GenyPluginConfigD
  - NameDecorator
  - Hw_TricoreCpu
  - DrvCan_TricoreMulticanHll
    - Individual Polling
    - Channels
      - DUT_CH0
        - Tx Messages
        - Rx Messages

	Object Type	Channel	Enable Polling
Normal Tx Polling	Normal Tx	DUT_CH0	□ *
LowLevel Tx Polling	Lowlevel Tx	DUT_CH0	□ *
TxMSG00000032_0	FullCAN Tx	DUT_CH0	□ *
TxMSG00000052_0	FullCAN Tx	DUT_CH0	□ *
RxMSG00000411_0	FullCAN Rx	DUT_CH0	□ *
RxMSG00000311_0	FullCAN Rx	DUT_CH0	□ *
BasicCAN0 Rx Polling	BasicCAN Rx	DUT_CH0	□ *
Normal Tx Polling	Normal Tx	DUT_CH1	□ *

Figure 10-3 Configuration of individual polling with GENy

Features	Explanation
Configuration Options	
Enable Polling	If checked, the associated mailbox will be handled in polling mode. This is only selectable, if the polling mode is configured to individual polling.



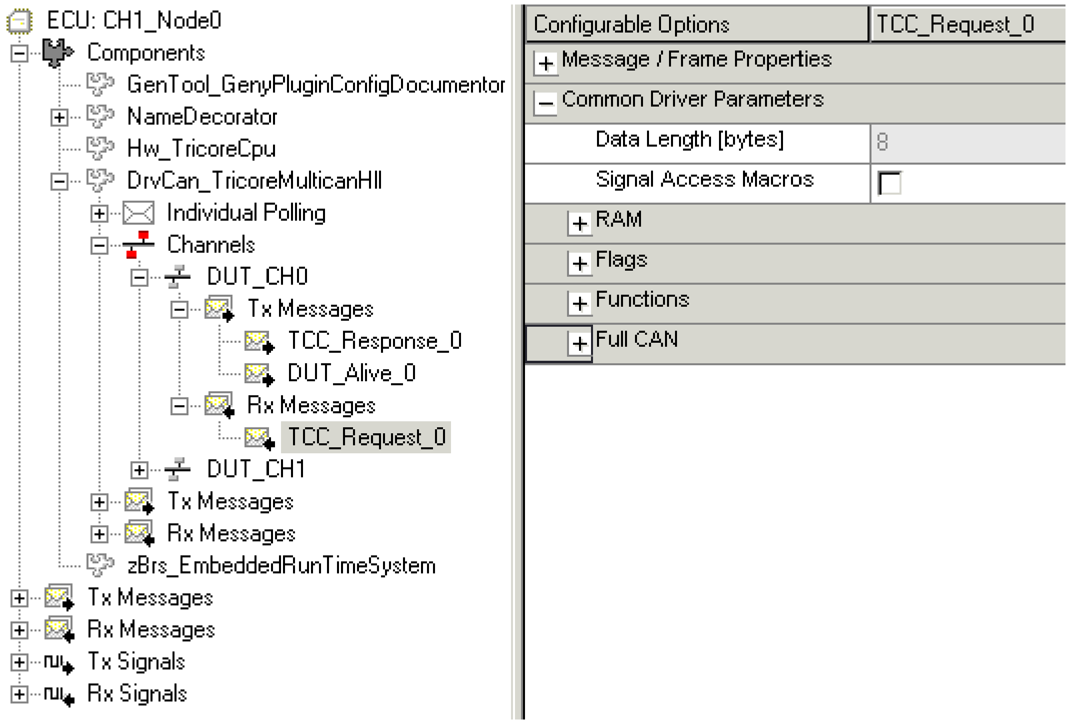
- ECU: CH1_Node0
  - Components
    - GenTool_GenyPluginConfigDocumentor
    - NameDecorator
    - Hw_TricoreCpu
    - DrvCan_TricoreMulticanHll
      - Individual Polling
      - Channels
        - DUT_CH0
          - Tx Messages
            - TCC_Response_0
            - DUT_Alive_0
          - Rx Messages
            - TCC_Request_0
        - DUT_CH1
      - Tx Messages
      - Rx Messages
    - zBrs_EmbeddedRunTimeSystem
  - Tx Messages
  - Rx Messages
  - Tx Signals
  - Rx Signals

Configurable Options	TCC_Response_0
+ Message / Frame Properties	
– Common Driver Parameters	
Signal Access Macros	□
+ Offline Modes	
+ RAM	
+ Flags	
+ Functions	
+ Full CAN Tx	

Figure 10-4 Configuration of a Tx message with GENy

Features	Explanation
Configuration Options	
Message / Frame Properties	
Generate	If unchecked, the generation of this message will be suppressed.
Common Driver Parameters	
Signal Access Macros	Signal access macros can be used by the application for an easy access to signals specified within a message.
Offline Modes	
<Part offline group>/<Mode X name>	
Usage	Standard: configuration via default mapping User Defined: the user can set or reset the “Real Value”
Real Value	It set, the message belongs to this group and the transmission of this message will be disabled if this group is switched offline.
Flags	
Confirmation Flag	After successful transmission of a message the driver sets the confirmation flag of the message.
Functions	
Confirmation Function	After successful transmission of a message the driver call the user defined confirmation function of the message. The name of this function has to be specified in this field.
Pretransmit Function	The used defined pretransmit function can be called before the transmission of a message is started. The name of this function has to be specified in this field.

Full CAN Tx	
Tx FullCAN	A Tx message can be assigned to a Full CAN objects.



ECU: CH1_Node0
- Components
  - GenTool_GenyPluginConfigDocumentor
  - NameDecorator
  - Hw_TricoreCpu
  - DrvCan_TricoreMulticanHll
    - Individual Polling
    - Channels
      - DUT_CH0
        - Tx Messages
          - TCC_Response_0
          - DUT_Alive_0
        - Rx Messages
          - TCC_Request_0
      - DUT_CH1
    - Tx Messages
    - Rx Messages
  - zBrs_EmbeddedRunTimeSystem
- Tx Messages
- Rx Messages
- Tx Signals
- Rx Signals

Configurable Options	TCC_Request_0
+ Message / Frame Properties	
− Common Driver Parameters	
Data Length [bytes]	8
Signal Access Macros	□
+ RAM	
+ Flags	
+ Functions	
+ Full CAN	

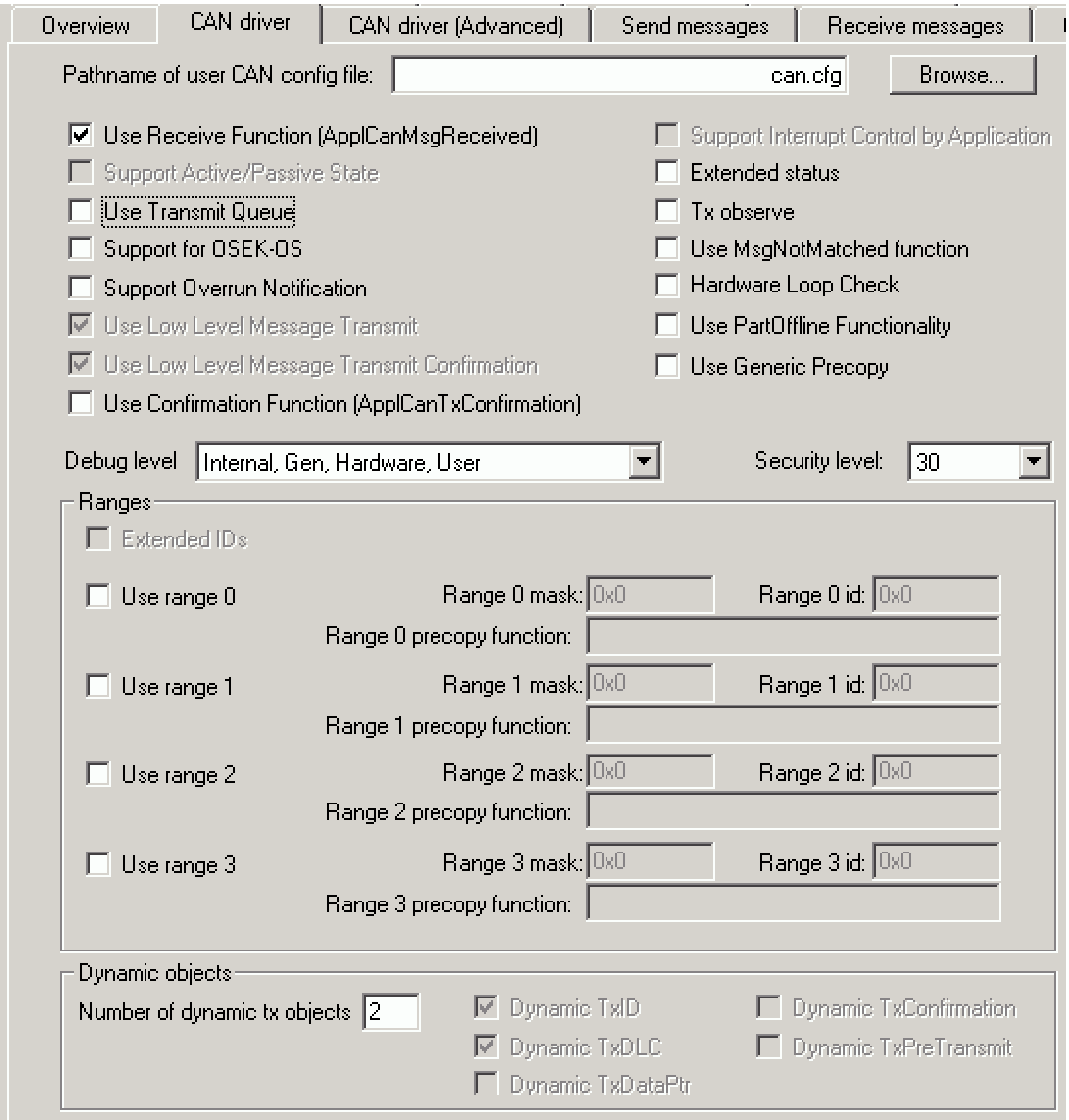
Figure 10-5 Configuration of an Rx message with GENy

Features	Explanation
Configuration Options	
Message / Frame Properties	
Generate	If unchecked, the generation of this message will be suppressed.
Common Driver Parameters	
Minimum Data Length	
Signal Access Macros	Signal access macros can be used by the application for an easy access to signals specified within a message.
Flags	
Indication Flag	After successful reception of a message the driver sets the indication flag of the message
Functions	
Indication Function	The name of the used defined indication function can be specified in this field.
Precopy Function	The name of the used defined precopy function can be specified in this field.
Full CAN	
Full CAN	An Rx message can be assigned to a Full CAN objects.
Lock Full CAN	If set, the assignment of a Rx message to a Full CAN

	object cannot be changes by the filter optimization.
Hardware Channel	In case a receive message is configured to be 'FullCAN' and Common CAN is activated then the user can configure this message to be received on the first (Channel A) or the second (Channel B) CAN controller on this channel.

10.3 Automatic Configuration by CANgen

Using the Generation Tool CANgen the configuration can be done by the tool. The configuration options common to all CAN Drivers is described here. The CAN Controller dependent options are described in the CAN Controller specific user manual. The following dialog describes the CAN Driver common options. The configuration data is stored by the tool in the file can_cfg.h for CANgen.



Pathname of user CAN config file: can.cfg

☑ Use Receive Function (ApplCanMsgReceived)
☐ Support Active/Passive State
☐ Use Transmit Queue
☐ Support for OSEK-OS
☐ Support Overrun Notification
☑ Use Low Level Message Transmit
☑ Use Low Level Message Transmit Confirmation
☐ Use Confirmation Function (ApplCanTxConfirmation)

☐ Support Interrupt Control by Application
☐ Extended status
☐ Tx observe
☐ Use MsgNotMatched function
☐ Hardware Loop Check
☐ Use PartOffline Functionality
☐ Use Generic Precopy

Debug level: Internal, Gen, Hardware, User
Security level: 30

Ranges

☐ Extended IDs

☐ Use range 0 — Range 0 mask: 0x0 — Range 0 id: 0x0 — Range 0 precopy function:
☐ Use range 1 — Range 1 mask: 0x0 — Range 1 id: 0x0 — Range 1 precopy function:
☐ Use range 2 — Range 2 mask: 0x0 — Range 2 id: 0x0 — Range 2 precopy function:
☐ Use range 3 — Range 3 mask: 0x0 — Range 3 id: 0x0 — Range 3 precopy function:

Dynamic objects

Number of dynamic tx objects: 2

☑ Dynamic TxID
☑ Dynamic TxDLC
☐ Dynamic TxDataPtr
☐ Dynamic TxConfirmation
☐ Dynamic TxPreTransmit

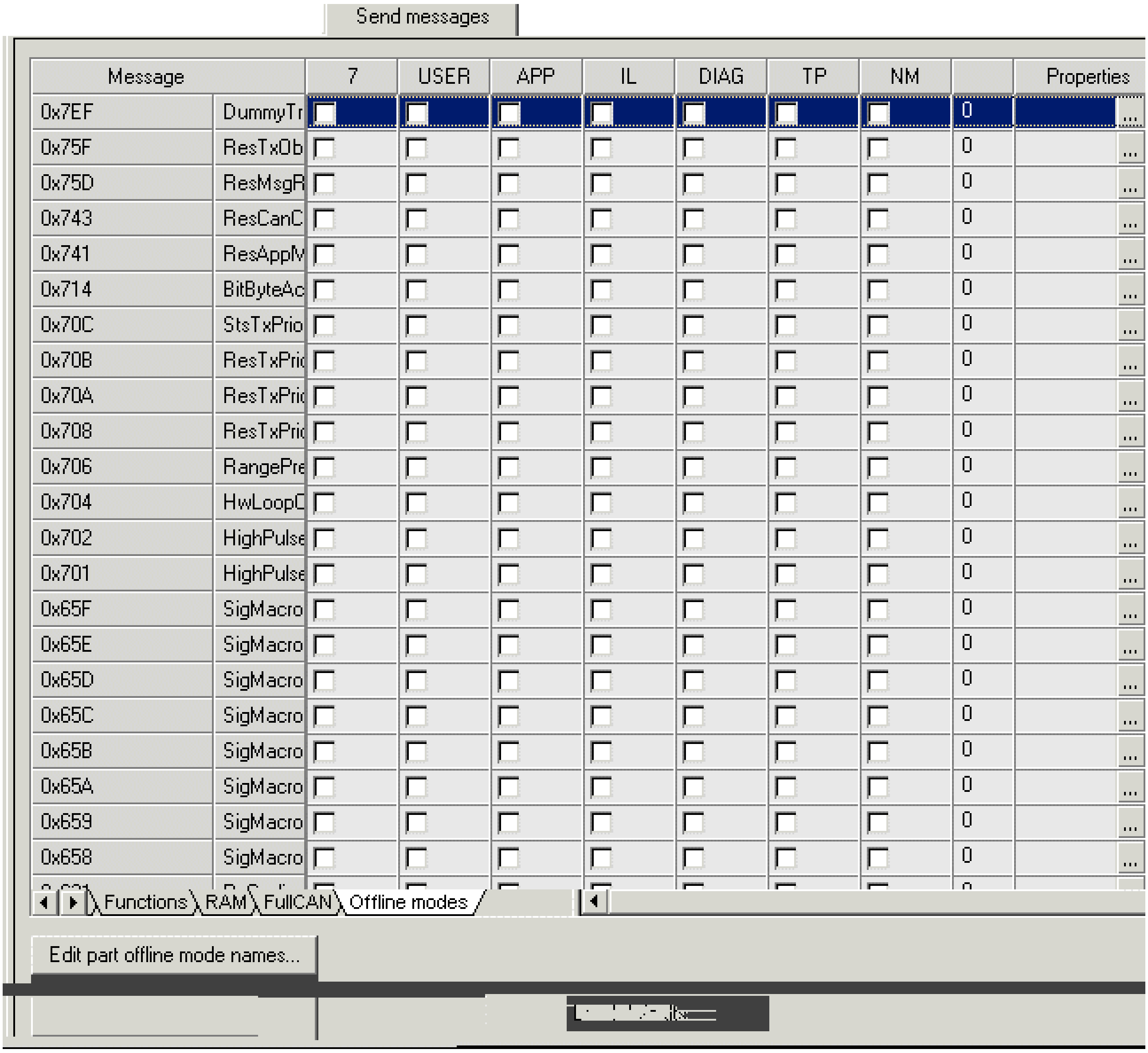
Figure 10-6 CAN Driver configuration tab

Feature	Explanation
Path of the CAN config file	The CAN Driver configuration file (can_cfg.h) is generated. If the user wants to overwrite this automatically generated configuration file, the user is able to define the name of a user defined configuration file which is included at the end of the generated file can_cfg.h. This means entries in the user defined configuration file overwrite the entries in can_cfg.h.
Use Receive Function (`ApplCanMsgReceived`)	The receive function `ApplCanMsgReceived()` is called by the CAN Driver on every reception of a CAN message after the hardware acceptance filter is passed. Within this callback function the Application may preprocess the received message in any way (ECU specific dynamic filtering mechanisms, preprocessing of the messages, gateway functionality...). If the callback function `ApplCanMsgReceived()` returns `kCanCopyData`, the CAN Driver continues to work on the message received. If the callback function returns `kCanNoCopyData`, the CAN finishes working on the message received.
Support Active/Passive State	If this field is checked, the CAN Driver's transmit path can be switched to the "Passive" state. In this state no CAN messages are sent to the CAN bus. The transmit request always returns with ”OK”. This ”passive” state functionality may be used to localize errors in a CAN bus: If there are errors in a CAN bus, and the errors disappear when a particular node is switched to passive state, the scapegoat is found. The Application must be switched to the passive state and back to active state by an external tester. If active/passive state is enabled, the CAN Driver is in active state after initialization.
Use Transmit Queue	If this field is checked the CAN Driver is configured to use a transmit queue. If no transmit queue is used, the Application is responsible to restart a transmit request if it wasn’t accepted by the CAN Driver. In the case of using a transmit queue, a transmit request is almost accepted. But the queue does only store the transmit request of a message. It doesn’t store the data to be sent in any case. The CAN Driver inserts a transmit request to the queue, if no hardware object is available when `CanTransmit()` is called. On a transmit interrupt, this means a former requested message is transmitted, the CAN Driver checks whether transmit requests are stored in the queue. If so, these requests are removed from the queue and the transmit request is executed. The search algorithm in the queue is priority based, there is no FIFO strategy. This means the identifier with the lowest number is removed first from the queue.
Support for OSEK OS	If this field is checked the CAN Driver is configured to support OSEK-OS. The kind of OSEK-OS depends on the specific microprocessor.
Use OsekOS Interrupt Cat 2	In case of using OSEK-OS the interrupt category of the CAN Driver interrupts have to be defined. Normally category 1 is used. Instead of this category 2 can be selected.

Feature	Explanation
Support Overrun Notification	If this field is checked the CAN Driver is configured to notify the Application in case of an overrun. The Application has to provide an Overrun callback function: void `ApplCanOverrun(void)` /* Overrun in the CAN Controller */ The overrun handling itself is done by the CAN Driver:
Security Level	This is the define value to configure the security level. Valid values are 0,10, 20 or 30.
Extended Status	This is the global checkbox for using hardware status information in the CAN Driver service `CanGetStatus()` or not.
Debug level	There are different Debug Levels supported by the CAN Driver: None: No debug functionality active. User: User API is debugged. The CAN Driver service function parameters are checked. Hardware: The CAN Controller interface is checked. Depends on CAN Controller. Gen: The configuration data are checked. Internal: CAN Driver internal checking (consistency of transmit queue).
Extended IDs	This checkbox is available only if extended CAN identifiers are selected in the channel configuration dialog. It has to be enabled if extended CAN identifiers have to be received by the range specific acceptance filtering and the appropriate precopy function has to be called. In such case no standard identifiers can be received by any acceptance range.
Use Range X	This is the global switch to select identifier range specific precopy functions. These ranges are normally used for Network Management, Transport Protocol and so on. There are in maximum 4 ranges configurable, where ‘X’ is the number of the specified range. If a range is enabled, the following additional settings has to be done:
Range X mask	Acceptance code of the identifier range X.
Range X ID	Acceptance mask of the identifier range X.
Range X precopy function	Specific precopy function for range X.
Tx observe	This is the global switch for using the tx observe functionality of the CAN Driver or not.
Use MsgNotMatched function	This is the global switch for using the `MsgNotMatched` function of the CAN Driver or not.
Hardware Loop Check	This is the global switch for using the hardware loop check of the CAN Driver or not.
Number of dynamic tx objects	Maximum number of dynamic send objects which are available at run time.
Dynamic TxId	If the checkbox “DynamicTxId” is checked, the IDs of the dynamic transmit objects can be changed by the service function `CanDynTxObjSetId()` and/or by the service function `CanDynTxObjSetExtId()`. The last mentioned service function is only available if extended ID addressing is checked in the Generation Tool.

Feature	Explanation
Dynamic TxDLC	If the checkbox “DynamicTxDLC” is checked, the DLCs of the dynamic transmit objects can be changed by the service function `CanDynTxObjSetDlc()`.
Dynamic TxDataPtr	If the checkbox “DynamicTxDataPtr” is checked, the data pointers of the dynamic transmit objects can be changed by the service function `CanDynTxObjSetDataPtr()`. The occurrence of this switch is CAN Controller dependent.
Dynamic TxConfirmation	If the checkbox “DynamicTxConfirmation” is checked, the confirmation function of the dynamic transmit objects can be changed by the service function `CanDynTxObjSetConfirmationFct()`. The occurrence of this switch is CAN Controller dependent.
Dynamic TxPreTransmit	If the checkbox “DynamicTxPretransmit” is checked, the pretransmit function of the dynamic transmit objects can be changed by the service function `CanDynTxObjSetPreTransmitFct()`.The occurrence of this switch is CAN Controller dependent.
Use Low Level Message Transmit	If the checkbox “Use Low Level Message Transmit” is checked, the function `CanMsgTransmit()` can be used.
Use Low Level Message Transmit Confirmation	This checkbox is only available, if “use Low Level Message Transmit” is active. The confirmation and init callback functions of low level transmit functionality can be activated by this checkbox.
Use PartOffline Functionality	If the checkbox “Use PartOffline Functionality” is checked, the partial offline mode is available.
Use Generic Precopy	This checkbox enables the use of the generic precopy function – `ApplCanGenericPrecopy()`. This precopy function is common to all receive messages. It will be called as soon as the receive handle is determined.

The next figure shows the Configuration of the partial offline mode. Every transmit message can be assigned to up to eight partial offline groups.



Message		7	USER	APP	IL	DIAG	TP	NM		Properties
0x7EF	DummyTr	□	□	□	□	□	□	□	0	...
0x75F	ResTxOb	□	□	□	□	□	□	□	0	...
0x75D	ResMsgR	□	□	□	□	□	□	□	0	...
0x743	ResCanC	□	□	□	□	□	□	□	0	...
0x741	ResAppM	□	□	□	□	□	□	□	0	...
0x714	BitByteAc	□	□	□	□	□	□	□	0	...
0x70C	StsTxPrio	□	□	□	□	□	□	□	0	...
0x70B	ResTxPric	□	□	□	□	□	□	□	0	...
0x70A	ResTxPric	□	□	□	□	□	□	□	0	...
0x708	ResTxPric	□	□	□	□	□	□	□	0	...
0x706	RangePre	□	□	□	□	□	□	□	0	...
0x704	HwLoopC	□	□	□	□	□	□	□	0	...
0x702	HighPulse	□	□	□	□	□	□	□	0	...
0x701	HighPulse	□	□	□	□	□	□	□	0	...
0x65F	SigMacro	□	□	□	□	□	□	□	0	...
0x65E	SigMacro	□	□	□	□	□	□	□	0	...
0x65D	SigMacro	□	□	□	□	□	□	□	0	...
0x65C	SigMacro	□	□	□	□	□	□	□	0	...
0x65B	SigMacro	□	□	□	□	□	□	□	0	...
0x65A	SigMacro	□	□	□	□	□	□	□	0	...
0x659	SigMacro	□	□	□	□	□	□	□	0	...
0x658	SigMacro	□	□	□	□	□	□	□	0	...

Figure 10-7Configuration of Partial Offline Mode

Feature	Explanation
Edit part offline mode names	this button can be used to change the names of the eight partial offline groups

The following features cannot be configured by the Application. They are set automatically depending on the used OEM:

- DLC check
- Data Copy Mechanism
- Cancel in Hardware

10.4 Manual configuration via user configuration file

This chapter describes additional configuration options for special features which can only be configured via user configuration file.

In the following table you will find a list of configuration switches, used to control the functional units of the CAN Driver:

Switch	Value / Range	Use of ...
C_xxx_APPLCANPREWAKEUP_FCT	ENABLE, DISABLE	Activate call of ApplCanPreWakeUp() if an WakeUp Interrupt occurs.
C_xxx_NOTIFY_CORRUPT_MAILBOX	ENABLE, DISABLE	Activate call of ApplCanCorruptMailbox() in case the CAN RAM Check fails for a certain mailbox.

If the Generation Tool CANgen is used, some additional configurations can only be due via user configuration file:

Switch	Value / Range	Use of ...
C_xxx_ONLINE_OFFLINE_CALLBACK_FCT	ENABLE, DISABLE	Activate call of ApplCanOnline() and ApplCanOffline() if the associated CAN driver state was entered.
C_xxx_INTCTRL_ADD_CAN_FCT	ENABLE, DISABLE	Activate call of `ApplCanAddCanInterruptDisable()` and `ApplCanAddCanInterruptRestore()`. These two functions have to be used to handle the wake-up interrupt if the hardware treats this interrupt separately or if the Driver runs in Polling Mode the polling tasks have to be disabled.

11 Glossary

Abbreviations and Expressions	Explanation
Acceptance filtering	Mechanism which decides whether each received protocol frame is to be taken into account by the local Node or ignored.
API	Application Program Interface.
Application Interface	An application interface is the prescribed method of a SW component for using the available functionality.
Arbitration	Mechanism which guarantees that a simultaneous access made by multiple stations results in a contention where one frame will survive uncorrupted.
ASAP	Arbeitskreis zur Standardisierung von Applikationssystemen. Standardization of Application and Calibration system task force
BCD	Binary Coded Decimal
Buffer	A buffer in a memory area normally in the RAM. It is an area, the application reserved for data storage
Bus	Defines what we call internal as channel or connection.
BusOff	A node is in the state bus off when it is switched off from the bus due to a request of fault confinement entity. In the bus off state, a node can neither send nor receive any frames. A node can start the recovery from bus off state only upon a user request.A node is in the state BusOff when it is switched off from the bus. In the state BusOff a node can neither send nor receive any protocol frames.
Callback function	This is a function provided by an application. E.g. the CAN Driver calls a callback function to allow the application to control some action, to make decisions at runtime and to influence the work of the Driver.
CAN	Controller Area Network protocol originally defined for use as a communication network for control applications in vehicles.
CAN Controller	A hardware unit integrated into a micro controller (or as a separate unit) handling the CAN protocol.
CAN Driver	The CAN driver encapsulates a specific CAN controller handling. It consists of algorithms for HW initialization, CAN message transmission and reception. The application interface supports both event and polling notification and WR/RD access to the message buffers.
Channel	A channel defines the assignment (1:1) between a physical communication interface and a physical layer on which different modules are connected to (either CAN or LIN). 1 channel consists of 1..X network(s).
Configuration	The communication configuration adapts the communication stack to the specific component requirements by means of the Generation Tool.
Confirmation	A service primitive defined in the ISO/OSI Reference model (ISO

	7498). With the service primitive 'confirmation' a service provider informs a service user about the result of a preceding service request of the service user. Notification by the CAN Driver on asynchronous successful transmission of a CAN message.
Data consistency	Data consistency means that the content of a given application message correlates unambiguously to the operation performed onto the message by the application. This means that no unforeseen sequence of operations may alter the content of a message hence rendering a message inconsistent with respect to its allowed and expected value.
DBC	CAN database format of the Vector company which is used by Vector tools
DLC	Data Length CodeNumber of data bytes of a CAN message
ECU	Electronic Control Unit
Error	Error is a local problem which could be solved locally. If not, the error will be given as an exception to the application. An error is not the specification conform misbehavior of a system (e.g. a not responded diagnostic request after three requests without response is no error). Discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition (IEC 61508-4).
FIFO	First In First Out
FILO	First In Last Out
Gateway	A gateway is designed to enable communication between different bus systems, e.g. from CAN to LIN.
Generation Tool	See CANgen, DBKOMGen and GENy. The generation tool configures the communication stack based on database attributes (vehicle manufacturer), project settings (module supplier) and license information (Vector).
HIS	Hersteller-Initiative Software
HW	Hardware
ID	Identifier (e.g. Identifier of a CAN message)
Indication	A service primitive defined in the ISO/OSI Reference Model (ISO 7498). With the service primitive 'indication' a service provider informs a service user about the occurrence of either an internal event or a service request issued by another service user. Notification of application in case of events in the Vector software components, e.g. an asynchronous reception of a CAN message.
Interrupt	Processor-specific event which can interrupt the execution of a current program section.
Interrupt level	Processing level provided for time-critical activities. To keep the interrupt latency brief, only absolutely indispensable actions should be effected in the Interrupt Service Routine, which ensures reception of the interrupt and trigger its (post) processing within a task. Other processing levels are: Operating System Level and Task Level.
ISO	International Standardization Organization

ISR	Interrupt Service Routine
LIN	Local Interconnect Network
Manufacturer	Vehicle manufacturer
Message	A message is responsible for the logical transmission and reception of information depending on the restrictions of the physical layer. The definition of the message contents is part of the database given by the vehicle manufacturer.
MISRA	Motor Industry Software Reliability Association
MRC	Multiple Receive Channel
Network	A network defines the assignment (1:N) between a logical communication grouping and a physical layer on which different modules are connected to (either CAN or LIN). 1 network consists of 1 channel, Y networks consists of 1..Z channel(s). We say network if we talk about more than 1 bus.
NM	Network Management
Node	A network topological entity. Nodes are connected by data links forming the network. Each node is separately addressable on the network.
OEM	Original Equipment Manufacturer
Offline	State of the data link layer. In the Offline state, no application communication is possible. Only the network management is allowed to communicate.
Online	(Normal) state of the data link layer. Application and Network Management communication are possible.
OS	Operating System
OSEK	Name of the overall project: Abbreviation of the German term "Offene Systeme und deren Schnittstellen fÃ¼r die Elektronik im Kraftfahrzeug" - Open Systems and the Corresponding Interfaces for Automotive Electronics.
Overrun	Overwriting data in a memory with limited capacity, e.g. Queued message object
Platform	The sum of micro controller derivative, communication controller implementation and compiler is called platform.
RAM	Random Access Memory
Register	A register is a memory area in the controller, e.g. in the CAN Controller. Distinguish Register from Buffer
RI	Reference Implementation
ROM	Read-Only Memory
Signal	A signal is responsible for the logical transmission and reception of information depending on the restrictions of the physical layer. The definition of the signal contents is part of the database given by the vehicle manufacturer. Signals describe the significance of the individual data segments within a message. Typically bits, bytes or words are used for data segments but individual bit combinations are also possible. In the CAN data base, each data segment is assigned a symbolic name, a value range, a conversion formula

	and a physical unit, as well as a list of receiving nodes.
SRC	Single Receive Channel
Status	A status describes the properties (parameters) of an entity. A state is interpreted as an information, e.g. an error, by the entity which uses a status, and is frequently determined by the history.
Task Level	Processing level where the actual application software, is executed. Tasks are executed according to the priority assigned to them, and to the selected scheduling policy. Other processing levels are: Interrupt level and Operating System Level.
Transceiver	A transceiver adapts the physical layer to the communication interface.
Vehicle Manufacturer	We use this instead of OEM
Watchdog	A monitoring entity.

12 Contact

Visit our website for more information on

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

www.vector-informatik.com