Vector CAN Driver

Technical Reference

Renesas

RH850

RSCAN

Version 1.05.00

Authors	Torsten Kercher
Status	Released

Document Information

History

Author	Date	Version	Remarks
Torsten Kercher	2013-05-27	1.00.00	Initial release (support F1L with Green Hills compiler)
Torsten Kercher	2013-07-18	1.01.00	Support R1L Correct description of nested interrupt behavior
Torsten Kercher	2013-08-26	1.02.00	Support HighEnd features Support Diab compiler
Torsten Kercher	2013-10-16	1.03.00	Support R1M Update referenced version of the R1x hardware manual Support external wakeup functionality Update chapters 5, 6, 7.2.2
Torsten Kercher	2014-04-04	1.04.00	Support extended CAN RAM check Support RSCAN RAM test Support D1L, D1M, P1M Update referenced version of the F1L hardware manual
Torsten Kercher	2014-04-29	1.04.01	Update description of nested interrupt behavior
Torsten Kercher	2014-05-15	1.05.00	Support IAR compiler Support F1H Update expected loop durations in chapter 5

Table 1-1 History of the document

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

Contents

Illustrations

Tables

1 Introduction

The concept of the CAN driver and the standardized interface between the CAN driver and the application is described in the document **TechnicalReference_CANDriver.pdf**. The CAN driver interface to the hardware is designed in a way that capabilities of the special CAN chips can be utilized optimally. The interface to the application was made identical for the different CAN chips, so that the "higher" layers such as network management, transport protocols and especially the application would essentially be independent of the particular CAN chip used.

This document describes the hardware dependent special features and implementation specifics of the Renesas RSCAN on the RH850 platform.

2 Important References

The following table summarizes information about the CAN Driver. It gives you detailed information about the versions, derivatives and compilers. As very important information the documentations of the hardware manufacturers are listed. The CAN Driver is based upon these documents in the given version.

Driver Version	Supported Compilers	Supported Derivatives	Hardware Manufacturer Document	Version
3.11.xx, RI 1.5	Green Hills, Wind River Diab, IAR	D1L D1M	R01UH0451EJ0041, RH850/D1L/D1M Group, User's Manual: Hardware	Rev.0.41 Jan 2014
		F1L	R01UH0390EJ0100, RH850/F1L Group, User's Manual: Hardware	Rev.1.00 Jan 2014
		F1H [1]	R01UH0445EJ0010, RH850/F1H Group, User's Manual: Hardware	Rev.0.10 Jan 2014
		P1M	R01UH0436EJ0041, RH850/P1x Group, User's Manual: Hardware	Rev.0.41 Nov 2013
		R1L R1M	R01UH0411EJ0061, RH850/R1x Group, User's Manual: Hardware	Rev.0.61 Aug 2013
			R01US0058EJ0020, RH850 Family, User's Manual: Software	Rev.0.20 Feb 2013

Table 2-1 Supported Hardware Overview

Driver Version: This is the current version of the CAN Driver. RI shows the version of the Reference Implementation and therefore the functional scope of the CAN Driver.
Supported Compilers: List of compilers the CAN Driver is working with.
Supported Derivatives: List of derivatives the CAN Driver can be used on.
Hardware Manufacturer Document: List of hardware documentation the CAN Driver is based on.
Version: To be able to reference to this hardware documentation its version is very important.

[1] Only RSCAN0 is supported (physical channels CAN0-CAN5).

3 Usage of Controller Features

3.1 [#hw_comObj] - Communication Objects

The generation tool supports a flexible allocation of message buffers:

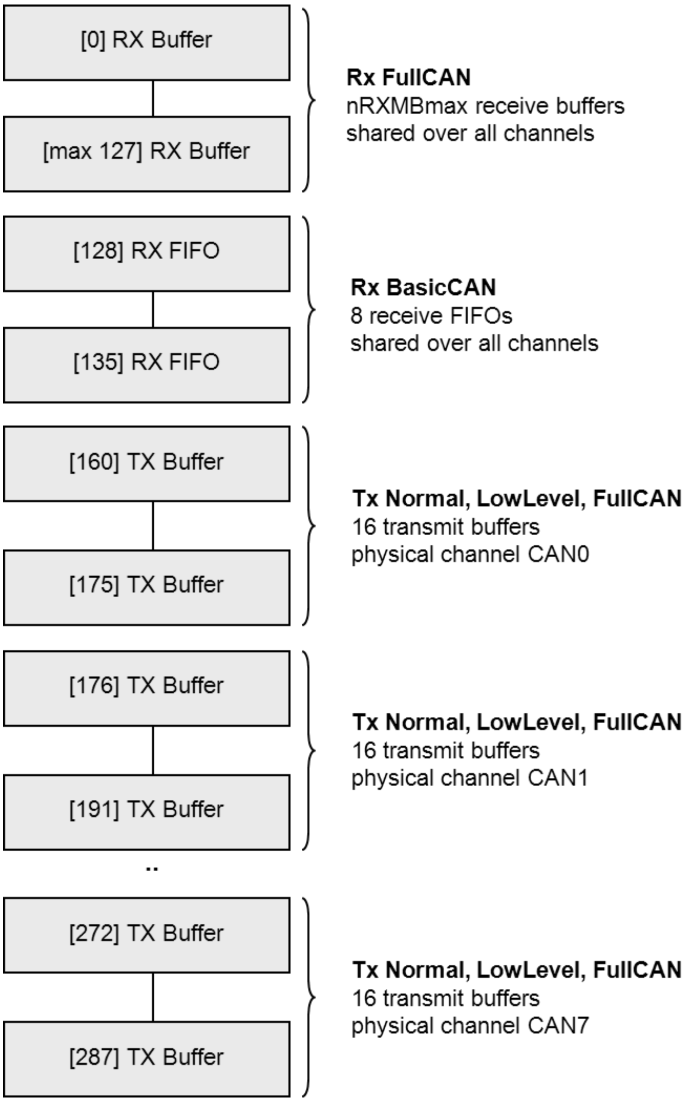


Figure 3-1 Hardware Object Layout

Note
Figure 3-1 depicts the maximum RSCAN capacities - the actual layout depends on the used derivative. Refer to the hardware manual to get the number of supported physical channels to determine which Tx buffers are available. The amount of supported Rx buffers (nRXMBmax) equals the number of supported physical channels * 16.

Obj number	Hw object type	Log object type	No. of objects	Comment
0 - (nRxFC -1)	Receive buffer	Receive FullCAN	0 - nRxMBmax = nRXFC	These objects are used to receive specific CAN messages. The user defines statically (Generation Tool) that a CAN message should be received in a FullCAN message object. The Generation Tool distributes the messages to the FullCAN objects. Up to nRxMBmax receive FullCAN objects can be configured per channel, but the sum over all receive FullCAN objects on all channels must not exceed nRxMBmax. The receive buffers for the FullCAN objects of all channels (sorted ascending by the physical channel index) are allocated continuously starting from index 0.
nRxFC - 127	Receive buffer	Unused	0 - 128	These objects are not used. It depends on the configuration of receive FullCAN objects and nRxMBmax how many receive buffers are not used. These objects will not be configured, so they don't consume shared hardware buffers.
128 - (128 + nRxBC -1)	Receive FIFO buffer	Receive BasicCAN	0 - 8 = nRXBC	All other CAN messages (Application, Diagnostics, Network Management) are received via the BasicCAN objects. Each object consists of one receive FIFO buffer with a configurable amount of acceptance filters and an individually configurable FIFO depth (number of allocated shared buffers). In general there is one BasicCAN object per channel, but by using the Multiple BasicCAN feature the amount of used BasicCAN objects can be configured. Up to 8 receive BasicCAN objects can be configured per channel, but the sum over all receive BasicCAN objects on all channels must not exceed 8. The receive FIFO buffers for the BasicCAN objects of all channels (sorted ascending by the physical channel index) are allocated continuously starting from index 128.
128 + nRxBC - 135	Receive FIFO buffer	Unused	0 - 8	These objects are not used. It depends on the configuration of receive BasicCAN objects how many receive FIFO buffers are not used. These objects will not be configured, so they don't consume shared hardware buffers.

Obj number	Hw object type	Log object type	No. of objects	Comment
136 - 159	Transmit / Receive FIFO buffer	Unused	24	The usage of Transmit / Receive FIFO buffers is not supported by this driver. These objects are always unused and will not be configured, so they don't consume shared hardware buffers.
160 + (n*16)	Transmit buffer	Transmit Normal	1 per channel	This object is used by `CanTransmit()` to send several messages on the logical channel that is mapped to physical channel n. If the transmit message object is busy, the transmit request is stored in a software queue.
161 + (n*16)	Transmit buffer	Low Level Transmit	0 or 1 per channel = nTXLL	This object is used by `CanMsgTransmit()` to send its messages on the logical channel that is mapped to physical channel n if the Low Level transmit functionality is used.
161 + (n*16) + nTXLL - 161 + (n*16) + nTXLL + nTXFC(n) -1	Transmit buffer	Transmit FullCAN	0 - 15 per channel = nTXFC(n)	These objects are used by `CanTransmit()` to send a certain message on the logical channel that is mapped to physical channel n. The user defines statically (Generation Tool) which CAN messages are located in such Tx FullCAN objects. The Generation Tool distributes the messages to the objects. Up to 15 transmit FullCAN objects can be assigned per channel (up to 14 if the Low Level transmit functionality is used).
161 + (n*16) + nTXLL + nTXFC(n) - 161 + (n*16) +14	Transmit buffer	Unused	0 - 15 per channel	These objects are not used. It depends on the configuration of transmit objects how many transmit buffer objects are not used. Additionally all transmit buffers of not supported or unused physical channels n are unused.

Table 3-1 Hardware Object Layout

nRxMBmax	Amount of RX buffers that is supported by the used derivative (see note above)
nRxFC	Number of used Rx FullCAN objects over all channels
nRxBC	Number of used Rx BasicCAN objects over all channels
n	Index of the physical channel
nTXLL	Number of Low Level transmit objects per channel (0 or 1)
nTXFC(n)	Number of used Tx FullCAN objects on the channel that is mapped to the physical channel n

Caution
The number of available transmit buffer objects per physical channel is constant. The receive buffers and FIFOs are shared over all channels and the availability per channel is restricted as explained in table 3-1. Furthermore the internal buffers for all receive objects are allocated out of a common buffer pool with size of (number of supported physical channels * 64). This has to be considered when configuring the number of the Rx FullCAN objects and the number and individual FIFO depths of the Rx BasicCAN objects (refer to section 11.1.3 for further information and details on how to configure the hardware objects).

3.2 Acceptance Filters

The hardware acceptance filters of the receive BasicCAN objects must allow reception of all messages that are not received in FullCAN message objects and additionally all messages that fit in a configured range (e.g. for Network Management, Transport Protocol). The generation tool offers assistance for configuration. The number of used filters is also configurable to allow efficient hardware filtering to minimize unnecessary CPU load.

Caution
The hardware supports a pool of acceptance filters with size of (number of supported physical channels * 64) that are used for Rx BasicCAN as well as Rx FullCAN objects. This has to be considered when configuring the number of Rx FullCAN objects and the number of filters per Rx BasicCAN object. See section 11.1.3 for further information and details on the configuration.

4 [#hw_sleep] - SleepMode and WakeUp

The driver supports sleep and wakeup functionality. With the function `CanSleep()` the CAN controller enters sleep mode and leaves it with the function `CanWakeUp()` (internal wakeup) or upon a falling edge respectively dominant level on the Rx pin (external wakeup).

If you want to use sleep/wakeup functionality specify the Sleep/Wakeup option in the generation tool. If this functionality is not configured, the service functions `CanSleep()` and `CanWakeUp()` are empty and return `kCanNotSupported`.

4.1 Sleep

The function `CanSleep()` changes the channel from communication mode via reset mode to stop mode. If the function is called during CAN communication, the reception or transmission is terminated before it is completed (the same applies to a call of `CanResetBusSleep()`).

These transitions do not depend on external influences (e.g. the CAN bus level), so the return value `kCanOk` is always expected. However, if the function returns `kCanFailed` call `CanSleep()` again or re-initialize the channel.

4.2 Internal Wakeup

The function `CanWakeUp()` changes the channel from stop mode via reset mode to communication mode.

The return value `kCanOk` is always expected for this mode change. However, as the transition to operation mode takes two CAN bit times on the corresponding channel, the hardware loop `kCanLoopEnterOperationMode` is implemented (see chapter 5). If the function returns `kCanFailed` (e.g. caused by a loop cancellation) call `CanWakeup()` again or re-initialize the channel.

4.3 External Wakeup

The external wakeup functionality is realized by external interrupts but fully handled by the CAN driver. The RSCAN itself does not provide any possibility of detecting bus activity if it is in stop mode. Instead the port configuration of the RH850 allows combining the CANn Rx pin with an external interrupt INTPn to be able to detect a CAN event even if the driver is in sleep mode. Refer to chapter 10 for implementation hints.

When the driver is in sleep mode and a CAN event is detected on the Rx pin a wakeup interrupt is generated. This event can also be detected by polling. The ISR or the polling task calls the application function `ApplCanPreWakeUp()` (if configured), changes the channel mode via reset mode to communication mode and calls `ApplCanWakeUp()`.

Caution

If the Sleep/Wakeup functionality is enabled via the configuration tool both the internal and external wakeup are available and the CAN driver expects that the Rx pin of each used CAN channel is linked with an external interrupt in context of the RH850 port configuration. Also the driver expects that each external interrupt source is assigned to the lowest possible interrupt channel (check the "INTC1 interrupt select register" if applicable).

If this configuration is not possible for the actual derivative (refer to the hardware documentation) or any respective external interrupt cannot be used exclusively by the driver (write accesses to the corresponding interrupt control register), the external wakeup functionality can't be used.

In this case Sleep/Wakeup functionality itself has to be disabled or the external wakeup has to be deactivated by adding following to the user configuration file. Then only the internal wakeup is possible and the driver does not wake up on bus activity.

```
#define C_ENABLE_EXTERNAL_WAKEUP_SUPPRESSION
```

Caution

The driver performs write accesses within the interrupt controller address space of the MCU. If wakeup processing is configured to interrupt the corresponding interrupt is enabled at successful sleep transitions and disabled at successful wakeup transitions. Additionally the corresponding interrupt request flag is cleared right before the interrupt is enabled; hence a wakeup event can only be detected after the sleep transition has been completed. Refer to section 10.3 if an exclusive write access to the interrupt control registers is not possible. Please note that the interrupt request flag also has to be cleared for polling configurations.

5 [#hw_loop] - Hardware Loop Check

In context of the feature Hardware Loop Check (see TechnicalReference_CANDriver, chapter Hardware Loop Check) this CAN Driver provides the following timer identifications.

If the common term clock is used below always consider the one with the lower frequency out of clkc and pclk. The RSCAN requires clkc to be less or equal pclk/2, thus clkc can be taken as a basis in general. Refer to the hardware manual for a description of the different clocks and hardware specifics.

Caution
Always significantly increase the given durations for the loop callout implementation to compensate additional software delays.

kCanLoopRamInit
This loop may be called within the function `CanInitPowerOn()` and is processed until the CAN RAM initialization after a MCU reset has finished. This is necessary as this initialization has to be completed before the RSCAN can be configured. As this loop is not called in channel context the channel parameter has to be ignored. The maximum expected duration to wait for the CAN RAM initialization starts from the time of the MCU reset and is device specific. Refer to the corresponding hardware manual (e.g. section RSCAN Setting Procedure – Initial Settings) to get the number of required clock cycles. If the loop is canceled try to call `CanInitPowerOn()` again or reset the MCU.

kCanLoopInit
This loop may be called within the function `CanInitPowerOn()`. As it is not called in channel context the channel parameter has to be ignored. The loop may be called multiple times within this function and the possible occurrences are as follows: • To protect the transition via global reset mode to global stop mode (only active if the RSCAN ECC callout is enabled). • To protect the transition to global reset mode. • To protect transitions and settings in context of the global test mode (only active if the RSCAN RAM test is enabled) • To protect the transition to channel reset mode for each active channel. • To protect the transition to global operation mode. The duration for each mode transition in this context is expected to be 10 clock cycles at highest. If any loop occurrence is canceled try to call `CanInitPowerOn()` again or reset the MCU.

kCanLoopBusOffRecovery
This channel dependent loop may be called in `CanInit()` if the RSCAN is currently in BusOff state and is processed until 11 consecutive recessive bits have been detected 128 times on the bus to ensure compliance to the BusOff recovery specification (see chapter 6). The maximum expected duration is 1408 CAN bit times on a recessive bus, 128 message times (including inter-frame space) on a communicative bus or any time if disturbances are present. There is no issue and nothing to do if the loop is canceled, but the specified BusOff recovery time may not be met.

kCanLoopEnterResetMode
This channel dependent loop may be called in `CanInit()` and is processed as long as the CAN cell does not enter channel reset mode. The maximum expected duration of the loop is three clock cycles. If the loop is canceled try to reinitialize the channel again or call `CanInitPowerOn()`.

kCanLoopEnterOperationMode
This channel dependent loop is always called in `CanInit()`, `CanStart()` and `CanWakeUp()` and is processed as long as the CAN cell does not enter channel operation mode. The maximum expected duration of the loop is two CAN bit times on the current channel. If the loop is canceled try to call `CanStart()` respectively `CanWakeUp()` again or reinitialize the channel.

kCanLoopRxFcProcess
This channel depended loop is always called in `CanFullCanMsgReceived()` and is processed as long as new messages are received by the current hardware object while copying a previously received message to a temporary software buffer. This ensures that always consistent and most recent data is indicated. It is expected that the loop iterates one or two times. Please note that if the loop iterates more than one time previously received messages of the current receive buffer are discarded without further notification as data consistency cannot be ensured. There is no issue and nothing to do if the loop is canceled, but the latest message is also discarded and `CanFullCanMsgReceived()` returns without indicating any message at all.

6 [#hw_busoff] - Bus off

In case of a BusOff event the controller automatically changes to stop mode on the respective channel. There is no automatic recovery as specified by ISO11898-1; the application has to restart communication following the description in the Technical Reference CAN driver, i.e. by calling `CanResetBusOffStart/-End()` which leads to a call of `CanInit()`. Please note that if `CanResetBusOffEnd()` is called before 11 consecutive recessive bits have been detected 128 times, `kCanLoopBusOffRecovery` is called within `CanInit()` (see chapter 5).

7 CAN Driver Features

7.1 [#hw_feature] - Feature List

		Standard	HighEnd
Initialization			
Power-On Initialization		■	■
Re-Initialization		■	■
Transmission			
Transmit Request		■	■
Transmit Request Queue		■	■
Internal data copy mechanism		■	■
Pretransmit functions		■	■
Common confirmation function		■	■
Confirmation flag		■	■
Confirmation function		■	■
Offline Mode		■	■
Partial Offline Mode		■	■
Passive-Mode		■	■
Tx Observe mode		■	■
Dynamic TxObjects	ID	■	■
	DLC	■	■
	Data-Ptr	■	■
Full CAN Tx Objects		■	■
Cancellation in Hardware		■	■
Low Level Message Transmit		-	■
Reception			
Receive function		■	■
Search algorithms	Linear	■	■
	Table	-	-
	Index	■	■
	Hash	■	■
Range specific precopy functions (min. 2, typ.4)		4	4
DLC check		■	■
Internal data copy mechanism		■	■
Generic precopy function		■	■
Precopy function		■	■
Indication flag		■	■
Indication function		■	■
Message not matched function		■	■

Overrun Notification		■	■
Full CAN overrun notification		-	-
Multiple Basic CAN		-	■
Rx Queue [2]		-	■
Bus off			
Notification function		■	■
Nested Recovery functions		■	■
Sleep Mode			
Mode Change		■	■
Preparation		■	■
Notification function		■	■
Special Features			
Status		■	■
Security Level		■	■
Assertions		■	■
Hardware loop check		■	■
Stop Mode		■	■
Support of OSEK operating system		■	■
Polling Mode	Tx	■	■
	Rx (FullCAN) [3]	■	■
	Rx (BasicCAN)	■	■
	Error	■	■
	Wakeup	■	■
Individual Polling [4]		-	■
Multi channel		■	■
Support extended ID addressing mode		■	■
Support mixed ID addressing mode		■	■
Support access to error counters		■	■
Copy functions		■	■
CAN RAM check [5]		■	■
Extended CAN RAM check [6]		■	■

Table 7-1 CAN Driver Functionality

[2] Consider that the Rx BasicCAN hardware FIFOs in combination with Rx BasicCAN polling might be a more efficient alternative to the Rx Queue in many configurations.
[3] Due to hardware limitations (no interrupt request can be generated for receive buffers) Rx FullCAN polling is mandatory if Rx FullCAN objects are configured.
[4] Due to hardware limitations (see note 3) all Rx FullCAN objects have to be polled.
[5] Due to hardware limitations (no write access to Rx objects) only supported for Tx objects.
[6] This feature is project specific and only available if explicitly ordered.

7.2 Description of Hardware-related Features

7.2.1 [#hw_status] - Status

If a status is not supported, the related macro always returns false.

Status	Support
`CanHwIsOk(state)`	■
`CanHwIsWarning(state)`	■
`CanHwIsPassive(state)`	■
`CanHwIsBusOff(state)`	■
`CanHwIsWakeup(state)`	■
`CanHwIsSleep(state)`	■
`CanHwIsStart(state)`	■
`CanHwIsStop(state)`	■
`CanIsOnline(state)`	■
`CanIsOffline(state)`	■

Table 7-2 CAN Status

7.2.2 [#hw_stop] - Stop Mode

The service function `CanStop()` calls `CanInit()` and then puts the CAN Controller into channel reset mode, where it is disconnected from the bus. If the function is called during CAN communication, the reception or transmission is terminated before it is completed. This mode can be left by calling `CanStart()`. Both transitions do not depend on external influences (e.g. the CAN bus level), so the return value `kCanOk` is always expected. However, if the functions return `kCanFailed` (e.g. caused by a hardware loop cancellation) call `CanStop()`, respectively `CanStart()` again or re-initialize the channel.

7.2.3 [#hw_int] - Control of CAN Interrupts

CAN interrupt locking is performed by modifying the interrupt request mask bits (MK) in the control registers of the appropriate sources directly within the interrupt controller address space of the MCU. Therefore the driver needs exclusive write access to all CAN related EI level interrupt control registers (ICn). If the sleep/wakeup functionality is enabled this may include the ICn of external interrupt sources (see chapter 4).

Since Rx FIFO interrupt and overrun (global error) cannot be enabled and disabled for every object individually they are disabled globally when the interrupts of at least one controller are disabled and enabled globally if the interrupts of no controller are disabled anymore.

All CAN related ICn are initialized and then modified by the driver during runtime (interrupt disable and restore). The priority level for the initialization can be selected via the configuration tool (all CAN interrupts must have the same priority), see section 11.1.3. Refer to chapter 10 for further information on CAN interrupts.

Caution
In standard configuration the driver needs **exclusive** write access to all CAN related EI level interrupt control registers. Refer to chapter 10 for further information and especially section 10.3 if an exclusive write access is not possible.

7.2.4 [#hw_cancel] - Cancel in Hardware

	Yes	No
Has the `CanTxTask()` to be called by the application to handle the canceled transmit request in the hardware?		■

Cancelling transmission of messages via `CanCancelTransmit()` or `CanCancelMessageTransmit()`:

	Yes	No
`ApplCanTxConfirmation()` is only called for transmitted messages, successfully canceled messages are not notified. That means the CAN driver is able to detect whether a message is transmitted even if the application has tried to cancel.	■	

7.2.5 Remote Frames

Remote Frames will not have any influence on the communication because they are not received due to hardware filtering.

7.2.6 CAN RAM Check

The CAN driver supports a check of the CAN mailboxes which is performed internally every time the function `CanInit()` is called. The CAN driver verifies that no used mailboxes are corrupt. A mailbox is considered corrupt if predefined patterns are written to the appropriate mailbox registers and read operations do not return the expected patterns. If a corrupt mailbox is found the function `ApplCanCorruptMailbox()` is optionally called to inform the application which mailbox is affected.

After the check of all mailboxes on the given channel the CAN driver calls the function `ApplCanMemCheckFailed()` if at least one corrupt mailbox was found. The application can control whether the CAN driver disables communication on the current channel or not by means of the return value of the call-back function. If the application has decided to disable the communication there is no possibility to enable the communication again until the next call of `CanInitPowerOn()`.

Caution
Due to hardware limitations (no write access for receive objects) the CAN RAM check is only supported for transmit mailboxes. Consider the behavioural differences of CAN RAM check when it is used in combination with the extended CAN RAM check feature.

The additional call-back function `ApplCanCorruptMailbox()` can only be activated via the user configuration file - the settings are listed below. In case no user config file is used (i.e. the mentioned switch is not defined) the feature is disabled.

Switch	Value	Description
C_ENABLE_NOTIFY_CORRUPT_MAILBOX		Activate call of `ApplCanCorruptMailbox()` in case the CAN RAM check fails for a certain mailbox.

7.2.7 Extended CAN RAM Check

The extended RAM check provides an additional check of the CAN cell control registers RAM with extended API and modified standard CAN RAM check and driver behaviour.

The RSCAN control registers are differentiated between registers that have to be written in global reset mode (afterwards referred to as “global registers”) or can also be written in channel reset mode (afterwards referred to as “channel registers”). Since the transition to global reset mode affects all channels, the global register RAM is checked only once within `CanInitPowerOn()`. If the global register RAM is considered corrupt a call-back function (see below) is issued to allow the application to control whether the driver initialization is proceeded or not.

The channel register and mailbox RAM check is performed within the function `CanInit()`. The registers RAM check disables the complete channel communication if at least one of the checked registers is considered corrupt. The mailbox RAM check (only available for Tx objects) disables corrupt mailboxes so that no transmission is possible on them. In both cases the appropriate call-backs (see below) are called to inform the application about the failures. Channels and mailboxes can be re-enabled by the application using the extended API. If any of the control registers check or the mailbox registers check fails the overall RAM check call-back `ApplCanMemCheckFailed()` is invoked.

More detailed information is given below; section 9.3 describes the API functions:

- If any of the global registers (e.g. global configuration registers, registers relating to the configuration of receive objects and receive rules) are considered corrupt the function `ApplCanGlobalMemCheckFailed()` is invoked within `CanInitPowerOn()`. If this function returns `kCanEnableCommunication` the initialization is continued ignoring the results of the check. If it returns `kCanDisableCommunication` the RSCAN is put back into global stop mode and `CanInitPowerOn()` returns without initializing the RSCAN. The check of the channel registers RAM is also not performed and `CanInitPowerOn()` has to be called again to be able to use the CAN functionality in this case (other CAN API functions must not be called until `CanInitPowerOn()` was executed completely). The check is performed with every call of this function.
- If any of the channel control registers are considered corrupt the function `ApplCanCorruptRegisters()` is called and the communication on the given channel is disabled. The CAN cell stays in stop mode whatever the general call-back function `ApplCanMemCheckFailed()` returns and the channel is disconnected from the Tx port pin.

- If a corrupt mailbox is found it is disabled by the driver and the call-back function `ApplCanCorruptMailbox()` is invoked (if this is enabled by the definition of C_ENABLE_NOTIFY_CORRUPT_MAILBOX). In this case (but only if no corrupt CAN control registers were found on the given channel) the application can decide whether the communication on the channel should be disabled using the return value of the function `ApplCanMemCheckFailed()`, but the mailbox stays disabled anyway.
- If the communication on a channel was disabled previously it can be re-enabled using the function `CanEnableChannelCommunication()`.
- Mailboxes that were disabled by mailbox RAM check can be re-enabled by the function `CanEnableMailboxCommunication()` (but only if the communication on the given channel is enabled).
- No mailbox or register RAM check is performed and no RAM check call-backs are invoked if `CanInit()` is called by `CanResetBusOffEnd()`. However, all previously disabled channels or mailboxes stay disabled.

Caution
The only way to re-enable channel or mailbox communication is to use the functions `CanEnableChannelCommunication()`, `CanEnableMailboxCommunication()` or `CanInitPowerOn()`.

The extended CAN RAM check feature needs the standard CAN RAM check functionality to be activated. The following settings have to be done in the user configuration file. In case no user config file is used (i.e. the mentioned switch is not defined) the feature is disabled. Please note that this is a project specific feature that might not be available and C_ENABLE_CAN_RAM_CHECK_EXTENDED has no effect in this case.

Switch	Value	Description
C_ENABLE_CAN_RAM_CHECK_EXTENDED		Activate the extended CAN RAM check feature.

7.2.8 RSCAN ECC Configuration

In context of the RSCAN RAM error detection and correction (ECC) the driver provides the additional call-back function `ApplCanEccConfiguration()` (see section 9.2) that is invoked by `CanInitPowerOn()` after the CAN RAM initialization is complete and before the RSCAN is configured while the cell is in global stop mode. This gives the application the possibility to configure the ECC behavior for the RSCAN. The driver offers no further support for this feature - any ECC configuration and handling has to be performed by the application.

The following settings have to be done in the user configuration file. In case no user config file is used (i.e. the mentioned switch is not defined) the feature is disabled.

Switch	Value	Description
C_ENABLE_ECC_CALLOUT		Activate call of `ApplCanEccConfiguration()`.

7.2.9 RSCAN RAM Test

The RSCAN provides a global test mode that enables the driver to perform a check of the entire internal RSCAN RAM that is not accessible during normal operation. This check is performed once within `CanInitPowerOn()`. Similar to the (extended) CAN RAM check the internal RAM is considered corrupt if predefined patterns are written to the appropriate RAM addresses and read operations do not return the expected patterns. If any corrupt bits are found the call-back function `ApplCanGlobalMemCheckFailed()` is invoked (see section 9.3). If this function returns `kCanEnableCommunication` the initialization is continued ignoring the results of the check. If it returns `kCanDisableCommunication` the RSCAN is put back into global stop mode and `CanInitPowerOn()` returns without initializing the RSCAN. `CanInitPowerOn()` has to be called again to be able to use the CAN functionality in this case (other CAN API functions must not be called until `CanInitPowerOn()` was executed completely). The RAM test is performed with every call of this function.

The size of RAM to be checked is given by 2432 bytes multiplied with the number of CAN channels that are supported by the used derivative (as configured in the configuration tool; see section 11.1.2). The test always starts at the first address but the size to be checked can be changed via user configuration (see below).

If this test is used in combination with the (extended) CAN RAM check the latter one is reduced in order to save runtime.

- The receive rule registers are omitted by the global register check within the function `CanInitPowerOn()`.
- The mailbox check is omitted if `CanInit()` is called out of `CanInitPowerOn()`.

The following settings have to be done in the user configuration file. In case no user config file is used (i.e. the first switch is not defined) the feature is disabled. The optional second switch is only evaluated if C_ENABLE_CAN_HW_RAM_CHECK is defined.

Switch	Value	Description
C_ENABLE_CAN_HW_RAM_CHECK		Activate the RSCAN RAM test feature.
C_ENABLE_CAN_HW_RAM_CHECK_SIZE	32 .. 65504 (bytes)	Instead of the default size as stated above the given number of bytes is checked by the RAM test. The value must be a multiple of 32 bytes and has to be valid for the used derivative (refer to the corresponding hardware manual).

Caution
Depending on the size of RAM to be checked, used compiler options, clock settings and others this check might take up to several milliseconds. Suggestion is to verify the runtime of `CanInitPowerOn()` in the actual project if this feature is enabled.

8 [#hw_assert] – Assertions

The driver implements no specific assertions with additional error codes.

9 API

9.1 Category

Single Receive Channels (SRC)	
■	A "Single Receive Channel" CAN Driver supports one CAN channel.)
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)`).

Table 9-1 API Category

9.2 RSCAN ECC Configuration

In context of the RSCAN ECC feature the application has to provide following callback function (see section 7.2.8 further information).

ApplCanEccConfiguration

Prototype	
Single Receive Channel	`void ApplCanEccConfiguration (void)`
Multiple Receive Channel	`void ApplCanEccConfiguration (void)`
Parameter	
–	–
Return code	
–	–
Functional Description	
This function is called by `CanInitPowerOn()` to allow the configuration of the RSCAN ECC functionality.	
Particularities and Limitations	
Only available if `C_ENABLE_ECC_CALLOUT` is defined.	

9.3 (Extended) CAN RAM Check

In context of the CAN RAM check feature the application has to provide following callback functions (see sections 7.2.6 and 7.2.7 for further information).

ApplCanMemCheckFailed

Prototype	
Single Receive Channel	`vuint8 ApplCanMemCheckFailed (void)`
Multiple Receive Channel	`vuint8 ApplCanMemCheckFailed (CanChannelHandle channel)`
Parameter	
CanChannelHandle channel	This parameter specifies the CAN channel on which the memory check is performed.
Return code	
vuint8	**`kCanEnableCommunication`** – Allow communication (see note in “Particularities and Limitations”). **`kCanDisableCommunication`** – Disable communication, no reception and no transmission is performed.
Functional Description	
This call-back function is invoked within `CanInit()` if the CAN driver has found at least one corrupt bit within the CAN mailboxes RAM or (if extended CAN RAM check is enabled) at least one corrupt bit within the channel control registers RAM. The application can decide whether the CAN driver allows further communication by means of the return value.	
Particularities and Limitations	
Call context: If the feature Extended CAN RAM check is deactivated this function is called on task level or within the BusOff interrupt; else only on task level. **Configuration**: The following setting must be active: C_ENABLE_CAN_RAM_CHECK **Important note:** If the optional feature “Extended CAN RAM check” is activated (C_ENABLE_CAN_RAM_CHECK_EXTENDED is defined) and the registers RAM check failed (call-back function `ApplCanCorruptRegisters()` was called for the given channel), the communication on the channel will be disabled, the CAN cell stays in stop mode and the return value of this function is ignored – the communication will NOT be allowed even if the return value is `kCanEnableCommunication`.	

ApplCanCorruptMailbox

Prototype	
Single Receive Channel	`void **ApplCanCorruptMailbox** (CanObjectHandle hwObjHandle)`
Multiple Receive Channel	`void **ApplCanCorruptMailbox** (CanChannelHandle channel, CanObjectHandle hwObjHandle)`
Parameter	
CanChannelHandle channel	This parameter specifies the CAN channel on which the memory check is performed.
CanObjectHandle hwObjHandle	This parameter specifies the index of the corrupt mailbox.
Return code	
–	–
Functional Description	
This function is called within `CanInit()` if the CAN driver has found a corrupt mailbox.	
Particularities and Limitations	
Call context: If the feature "Extended CAN RAM check" is deactivated this function is called on task level or within the BusOff interrupt; else only on task level. **Configuration**: The following settings must be active: C_ENABLE_CAN_RAM_CHECK C_ENABLE_NOTIFY_CORRUPT_MAILBOX	

In case the feature extended CAN RAM check is enabled the following additional callback functions have to be provided by the application.

ApplCanCorruptRegisters

Prototype	
Single Receive Channel	`void **ApplCanCorruptRegisters** (void)`
Multiple Receive Channel	`void **ApplCanCorruptRegisters** (CanChannelHandle channel)`
Parameter	
CanChannelHandle channel	This parameter specifies the CAN channel on which the memory check is performed.
Return code	
–	–
Functional Description	
This function is called if the CAN driver has found corrupt channel control registers.	
Particularities and Limitations	
Call context: This function is called out of task level within `CanInit()` on the given channel if the RAM check is not suppressed. The RAM check is suppressed if `CanInit()` is called in scope of the functions `CanResetBusOffEnd()` or (dependent on parameter) `CanEnableChannelCommunication()`. **Configuration**: The following settings must be active: C_ENABLE_CAN_RAM_CHECK C_ENABLE_CAN_RAM_CHECK_EXTENDED	

ApplCanGlobalMemCheckFailed

Prototype	
Single Receive Channel	`vuint8` **`ApplCanGlobalMemCheckFailed`** `(void)`
Multiple Receive Channel	`vuint8` **`ApplCanGlobalMemCheckFailed`** `(void)`
Parameter	
–	–
Return code	
vuint8	**`kCanEnableCommunication`** – Continue initialization of the RSCAN. **`kCanDisableCommunication`** – Stop initialization of the RSCAN.
Functional Description	
This call-back function is invoked if the CAN driver has found at least one corrupt bit within the global control registers RAM in context of the extended CAN RAM check or if any corrupt bit was found in context of the RSCAN RAM test (see section 7.2.9). The application can decide whether the CAN driver proceeds with the RSCAN initialization by means of the return value.	
Particularities and Limitations	
Call context: This function is called out of task level within `CanInitPowerOn()`. **Configuration**: The following settings must be active: C_ENABLE_CAN_RAM_CHECK C_ENABLE_CAN_RAM_CHECK_EXTENDED or C_ENABLE_CAN_HW_RAM_CHECK **Important note:** Be aware of undefined runtime behavior if `kCanEnableCommunication` is returned as the driver tries to initialize and communicate despite corrupt RAM was found. The application has to verify the system in this case.	

The following service functions are provided by the driver in context of the extended CAN RAM check feature.

CanEnableChannelCommunication

Prototype	
Single Receive Channel	`void` **`CanEnableChannelCommunication`** `(vuint8 suppressRamCheck)`
Multiple Receive Channel	`void` **`CanEnableChannelCommunication`** `(CanChannelHandle channel, vuint8 suppressRamCheck)`
Parameter	
CanChannelHandle channel	This parameter specifies the CAN channel that shall be re-enabled.
vuint8 suppressRamCheck	**`kCanTrue`** – RAM check will be suppressed while re-enabling the communication on the channel. **`kCanFalse`** – RAM check will be performed while re-enabling the communication on the channel
Return code	
–	–
Functional Description	
The function re-enables the channel communication if it was disabled previously. It calls `CanInit()` internally but all eventually disabled mailboxes stay disabled. If the RAM check is not suppressed it can fail again and the appropriate call-back function is invoked in this case.	
Particularities and Limitations	
Restriction: Same restrictions as for a call of `CanInit()` apply. **Call context**: This function is called by the application. **Configuration**: The following settings must be active: C_ENABLE_CAN_RAM_CHECK C_ENABLE_CAN_RAM_CHECK_EXTENDED	

CanEnableMailboxCommunication

Prototype	
Single Receive Channel	`vuint8` **`CanEnableMailboxCommunication`** `(CanObjectHandle hwObjHandle)`
Multiple Receive Channel	`vuint8` **`CanEnableMailboxCommunication`** `(CanChannelHandle channel, CanObjectHandle hwObjHandle)`
Parameter	
CanChannelHandle channel	This parameter specifies the CAN channel for which the mailbox shall be re-enabled.
CanObjectHandle hwObjHandle	The index of the mailbox to be re-enabled.
Return code	
vuint8	**`kCanOk`** – Mailbox communication was re-enabled. **`kCanFailed`** – Enabling of mailbox communication failed: `hwObjHandle` is not a valid Tx mailbox, the mailbox was not disabled previously or the communication on the channel is still disabled.
Functional Description	
The function re-enables the mailbox communication that was disabled previously by the extended CAN RAM check. Note that the mailbox RAM check is not performed in scope of this function call - the application must guarantee that the mailbox is not corrupt.	
Particularities and Limitations	
Call context: This function is called by the application. **Configuration**: The following settings must be active: C_ENABLE_CAN_RAM_CHECK C_ENABLE_CAN_RAM_CHECK_EXTENDED	

9.4 CAN Interrupt Handling by Application

These additional call-back functions are used if the driver does not perform CAN interrupt handling. See section 10.3 for details and the functional description.

OsCanCanInterruptDisable

Prototype	
Single Receive Channel	`void OsCanCanInterruptDisable (void)`
Multiple Receive Channel	`void OsCanCanInterruptDisable (CanChannelHandle channel)`
Parameter	
CanChannelHandle channel	This parameter specifies the CAN channel for which the interrupts shall be disabled.
Return code	
–	–
Functional Description	
This function is called by `CanCanInterruptDisable()`.	
Particularities and Limitations	
Only available if `C_ENABLE_OSEK_CAN_INTCTRL` is defined.	

OsCanCanInterruptRestore

Prototype	
Single Receive Channel	`void OsCanCanInterruptRestore (void)`
Multiple Receive Channel	`void OsCanCanInterruptRestore (CanChannelHandle channel)`
Parameter	
CanChannelHandle channel	This parameter specifies the CAN channel for which the interrupts shall be restored.
Return code	
–	–
Functional Description	
This function is called by `CanCanInterruptRestore()`.	
Particularities and Limitations	
Only available if `C_ENABLE_OSEK_CAN_INTCTRL` is defined.	

ApplCanWakeupInterruptDisable

Prototype	
Single Receive Channel	`void ApplCanWakeupInterruptDisable (vuint8 channel)`
Multiple Receive Channel	`void ApplCanWakeupInterruptDisable (vuint8 channel)`
Parameter	
vuint8 channel	This parameter specifies the CAN channel for which the wakeup interrupt shall be disabled.
Return code	
–	–
Functional Description	
This function is called by `CanWakeup()` and `CanInit()`.	
Particularities and Limitations	
Only available if `C_ENABLE_OSEK_CAN_INTCTRL` and `C_ENABLE_SLEEP_WAKEUP` are defined and external wakeup is used.	

ApplCanWakeupInterruptEnable

Prototype	
Single Receive Channel	`void ApplCanWakeupInterruptEnable (vuint8 channel)`
Multiple Receive Channel	`void ApplCanWakeupInterruptEnable (vuint8 channel)`
Parameter	
vuint8 channel	This parameter specifies the CAN channel for which the wakeup interrupt shall be enabled.
Return code	
–	–
Functional Description	
This function is called by `CanSleep()`.	
Particularities and Limitations	
Only available if `C_ENABLE_OSEK_CAN_INTCTRL` and `C_ENABLE_SLEEP_WAKEUP` are defined and external wakeup is used.	

10 Implementations Hints

10.1 Important Notes

1. The following condition will lead to an endless recursion in the CAN Driver: Recursive call of 'CanTransmit' within a confirmation routine, if the CAN Driver has been set into the passive state by `CanSetPassive`. Recommendations are:
   - NO CALL OF `CanTransmit` WITHIN CONFIRMATION-ROUTINES
   - PLEASE USE `CanSetPassive` ONLY ACCORDING TO THE DESCRIPTION
2. Only the transmit line of the CAN Driver is blocked by the functions `CanOffline()`. However, messages in the transmit buffer of the CAN-Chip, are still sent. For a reliable prevention of this fact, call function `CanInit()` after calling `CanOffline()`. The order of the two function calls is urgently required, due to the fact, that `CanInit()` is only allowed in offline mode.
3. If the VStdLib interrupt-lock-level is used, the chosen priority level must be higher than the highest level of any functionality of the CAN Driver (signal access, etc). Keep in mind that smaller values represent higher priorities.
4. Resetting indication flags and confirmation flags is done by Read-Modify-Write. The application is responsible for consistency. `CanGlobalInterruptDisable()` and `CanGlobalInterruptRestore()` must be called to avoid interruption by the CAN. Otherwise confirmations or indications can be lost.
5. Port and general clock settings are not handled by the driver. This has to be performed by the upper layers before the call of `CanInitPowerOn()`. Please check the appropriate hardware manual of the used derivative for details regarding the hardware specific configuration aspects. The CAN clock (f_{CAN}) for baudrate generation can be selected via the configuration tool; refer to section 11.1.2.
6. If external wakeup support is used the port configuration (performed by the upper layers) has to be extended. Besides setting the correct port functions for CAN it has to be ensured that this function is combined with the respective external interrupt. Additionally the edge/level detection has to be configured correctly to generate interrupt requests upon detection of CAN events (e.g. on low level or falling edges) on the corresponding pins (see the hardware manual for details; refer to the filter control register for instance). If the external wakeup is used the control registers of the external interrupts are also fully handled by the CAN driver in default configuration.
7. When using Green Hills or IAR compiler the ID bit of the PSW is cleared by software when any category 1 interrupt service routine of the CAN driver is entered to allow nesting of interrupts. For other compilers the default platform behavior is not modified (the ID bit stays set) and the acknowledgement of further interrupt requests is blocked when any driver ISR is processed. This default driver behavior for category 1 interrupts can be changed by definition of `C_DISABLE_NESTED_INTERRUPTS` respectively `C_ENABLE_NESTED_INTERRUPTS` via the user configuration file. Keep in mind that the feature is redundant if the compiler inserts code to allow nesting of interrupts in general and always verify that the compiler generates correct code if the feature is enabled.

10.2 Interrupt Configuration

With exception of the CAN related EI level interrupt control registers (ICn, see section 7.2.3) all further interrupt configuration within the interrupt controller address space of the MCU has to be performed by the application before the call of `CanInitPoweron()`.

The default implementation configures table reference as the way to determine the interrupt vector (TB bit in ICn registers is set). The application has to take care about referencing the CAN interrupt service routines in the interrupt vector table - the prototypes are exported in the driver header file. Please check the appropriate hardware manual of the used derivative for details regarding the hardware specific configuration aspects. Table 10-1 shows the provided ISRs and the accordant interrupt sources (n is the index of the physical channel). Please note that it is configuration dependent whether a particular interrupt service routine is available (see remarks in table).

CAN interrupt request name	CAN interrupt request cause	Provided service routine	Remarks
INTRCANGRECC	CAN RX FIFO interrupt	CanIsrRxFifo	Used for BasicCAN reception if ‘Rx BasicCan Polling’ is not enabled or ‘Individual Polling’ is configured.
INTRCANGERR	CAN global error interrupt	CanIsrGlobalStatus	Used for Rx BasicCAN overrun if ‘Error Polling’ is not enabled.
INTRCANnTRX	CANn TX interrupt	CanIsrTx_n	Used for transmission on physical channel n if ‘Tx Polling’ is not enabled or ‘Individual Polling’ is configured.
INTRCANnREC	CANn TX/RX FIFO RX complete interrupt	-	This interrupt is not used.
INTRCANnERR	CANn error interrupt	CanIsrStatus_n	Used for BusOff detection on physical channel n if ‘Error Polling’ is not enabled.
INTPn	External interrupt (see chapter 4)	CanIsrWakeup_n	Used for wakeup detection on physical channel n if the sleep/wakeup functionality is enabled, the external wakeup is used and ‘Wakeup Polling’ is not enabled.

Table 10-1 Interrupt Service Routines

If the INTC shall implement direct jumps to an address determined by the interrupt priority level (instead of table reference) the switch `C_ENABLE_DIRECT_INTERRUPT_BRANCH` has to be defined via the user configuration file. (This setting affects the TB bit in the ICn registers.) In this case the application has to implement a common service routine for all CAN interrupts and jump to it from the corresponding address (refer to hardware manual for configuration aspects).

See below an implementation example for Green Hills compiler and a full interrupt system with disabled sleep/wakeup functionality that uses physical channels 1 and 4; also refer to the information in table 10-1 about the presence of the individual CAN interrupt functions. Each driver routine must not be called if the CAN interrupts for the corresponding channel (respectively any CAN channel for the global handlers) are currently disabled. This is especially relevant if more than one channel is used or other interrupt sources also call the common service routine. In general it is recommended to check the status of the MK bit in the ICn register of each CAN interrupt source before invoking the corresponding driver routine as these bits directly indicate the status of the CAN interrupt lock mechanism. Any driver routine may only be called if the corresponding interrupt source is enabled (MK bit == 0). These actions may differ if the application handles the CAN interrupt disable/restore mechanism (see section 10.3 below), but the requirements above must always be met. If the feature “Multiple Configurations” is used only functions corresponding to channels that are used in the active identity should be called.

```
#pragma ghs interrupt
void CommonIsr_Prio_x ( void )
{
  /* handling for other interrupts that are assigned to
     this priority and not handled by table reference   */

  /* CAN interrupts */
  if (MK bit of INTRCANGRECC == 0) CanIsrRxFifo();
  if (MK bit of INTRCANGERR == 0) CanIsrGlobalStatus();
  if (MK bit of INTRCAN1ERR == 0) CanIsrStatus_1();
  if (MK bit of INTRCAN1TRX == 0) CanIsrTx_1();
  if (MK bit of INTRCAN4ERR == 0) CanIsrStatus_4();
  if (MK bit of INTRCAN4TRX == 0) CanIsrTx_4();

  /* handling for other interrupts that are assigned to
     this priority and not handled by table reference   */
}
```

Since the common service routine is already qualified as an ISR to the compiler, the individual CAN interrupt routines have to be configured as void-void functions if this variant is used. Therefore the switch `C_ENABLE_ISRVOID` additionally has to be defined via the user configuration file (if category 1 CAN interrupts are used).

Caution

The driver performs no measures to ensure the correct functionality of the CAN interrupt disable/restore mechanism if it is bypassed by the common interrupt handler when `C_ENABLE_DIRECT_INTERRUPT_BRANCH` is defined. Therefore the usage of this switch in not recommended in general and should only be defined if table reference is not possible at all.

10.2.1 Configuration of Interrupt Vectors with IAR compiler

Instead of a manual initialization of the interrupt vector table it is possible to let the IAR compiler set up the table by using the #pragma vector=xx directive (only for category 1 interrupts). This feature can be enabled via the user configuration file by defining the EI level interrupt number for each used CAN interrupt. The names of these defines are derived from the corresponding ISR names.

See below an example for a full interrupt system with external wakeup support that uses physical channels 2 and 5. Refer to the information in table 10-1 about the presence of the individual CAN interrupt functions.

```
#define C_CANISRRXFIFO_VECTOR         15
#define C_CANISRGLOBALSTATUS_VECTOR   14
#define C_CANISRTX_VECTOR_2           211
#define C_CANISRSTATUS_VECTOR_2       209
#define C_CANISRWAKEUP_VECTOR_2       31
#define C_CANISRTX_VECTOR_5           281
#define C_CANISRSTATUS_VECTOR_5       279
#define C_CANISRWAKEUP_VECTOR_5       36
```

Caution
This is an example and the necessary defines depend on the actual configuration. The interrupt numbers depend on the selected derivative; refer to the hardware manual to get the respective values.

10.3 CAN Interrupt Handling by Application

If an exclusive write access to the CAN related EI level interrupt control registers (ICn) is not possible or if the driver must not access registers outside the CAN address space the switch `C_ENABLE_OSEK_CAN_INTCTRL` has to be defined via the user configuration file. In this case the driver does not access the registers of the interrupt controller at all and the application has to ensure proper initialization, disabling and restoring of the CAN interrupt sources.

The initialization of all necessary ICn has to be performed additionally by the application before the call of `CanInitPowerOn()` if this switch is defined. All used sources (see remarks in table 10-1) have to be enabled after initialization whereas unused sources have to be disabled.

In context of the interrupt disable/restore mechanism the driver implements application call-backs that are used whenever the functions `CanCanInterruptDisable()` or `CanCanInterruptRestore()` are invoked (see section 9.4 for API definitions). The function `OsCanCanInterruptDisable()` should save the current mask status (MK bits) of all used CAN interrupt sources that are linked to the given logical channel and then set these bits to 1 in order to disable the sources. `OsCanCanInterruptRestore()` has to restore the previously saved mask bits for the given logical channel. These actions may differ based on the actual software configuration, but all CAN interrupts on the corresponding channel have to be locked after the first call (nested calls can occur) of `OsCanCanInterruptDisable()` and be available not until the last nested call of `OsCanCanInterruptRestore()`. Keep in mind that the right physical channel has to be chosen based on the given logical controller (to get the right ICn) and that the receive FIFO and global status interrupt are used by all controllers.

If sleep/wakeup functionality is enabled and the switch `C_ENABLE_OSEK_CAN_INTCTRL` is defined please note that the external wakeup interrupts always have to be disabled after initialization as these sources are only enabled on demand. Also there are two additional application call-back functions. See section 9.4 for API definitions. This is relevant for interrupt and polling systems as the external interrupt request flag has to be cleared independently of the interrupt configuration.

`ApplCanWakeupInterruptEnable()` is always invoked in context of `CanSleep()`. This function first has to clear the corresponding external interrupt request flag in the ICn and then – only if wakeup processing is performed by interrupts – enable the external interrupt. Keep in mind that depending on the current status of the CAN interrupt disable/restore mechanism this has to be performed either by clearing the corresponding mask bit in the respective hardware register or in the mask status that was saved by `OsCanCanInterruptDisable()`.

`ApplCanWakeupInterruptDisable()` is only invoked if wakeup processing is performed by interrupts in context of `CanWakeup()`, respectively the external wakeup handling and in `CanInit()`. This function has to disable the interrupt - depending on the current status of the CAN interrupt disable/restore mechanism either by setting the corresponding mask bit in the hardware register or in the saved mask status.

Caution

If `C_ENABLE_OSEK_CAN_INTCTRL` is defined the driver performs no measures to ensure consistency of the interrupt lock mechanism. Additionally the application has to ensure correct concurrent accesses to the ICn by all modules and has to handle nested calls of `OsCanCanInterruptDisable()` and `OsCanCanInterruptRestore()`.Therefore the usage of this switch is not recommended in general and should only be defined if the internal driver mechanisms are not possible at all (e.g. write access to interrupt control registers is not allowed).

11 Configuration

11.1 Configuration by GENy

The driver is configured with the help of the configuration tool GENy. This section describes the configuration of the driver specific aspects. The configuration options common to all CAN drivers are described in TechnicalReference_CANDriver.pdf.

Note
To get further information please refer to the online help of the generation tool.

11.1.1 Platform Settings

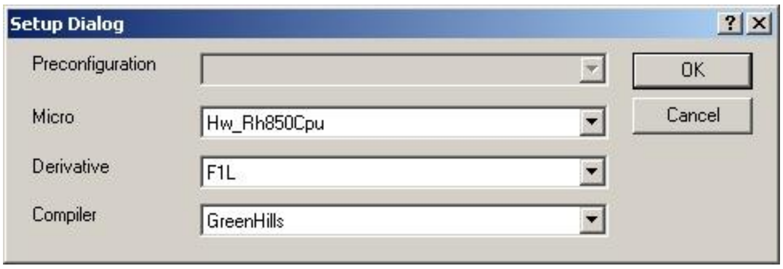


Figure 11-1 GENy Platform Settings

Attribute	Supported Values	Description
Preconfiguration		Select the pre-configuration file to use.
Micro	Hw_Rh850Cpu	Select the target platform.
Derivative	See Table 2-1	Select the specific derivative group.
Compiler	See Table 2-1	Select the used compiler.

Table 11-1 GENy Platform Settings

11.1.2 Component Settings

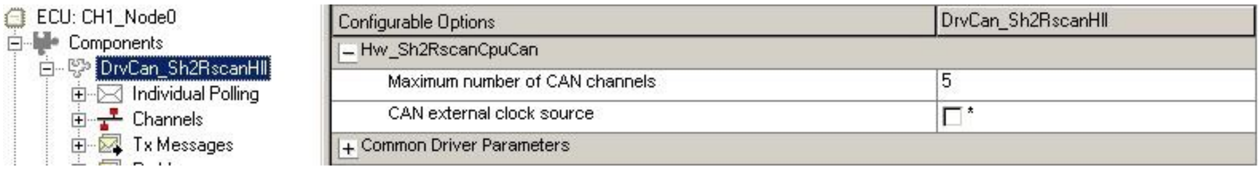


Figure 11-2 GENy Component Settings

Attribute	Supported Values	Description
Maximum number of CAN channels	1 - 8	Enter the maximum number of physical CAN channels that are supported by the actually used derivative. This value is independent from the number of channels in the configuration but used to determine the available hardware resources. Only if this value is correct the tool can ensure valid configurations for the actual derivative. Depending on the selected derivative not all values may be available.
CAN external clock source	true, false	Enable this attribute to use the external clock input (clk_xincan) as CAN clock (f_{CAN}). If the attribute is disabled clkc is used - this is the default selection. (This setting directly affects the DCS bit in the global configuration register.)

Table 11-2 GENy Component Settings

11.1.3 Channel-specific Settings

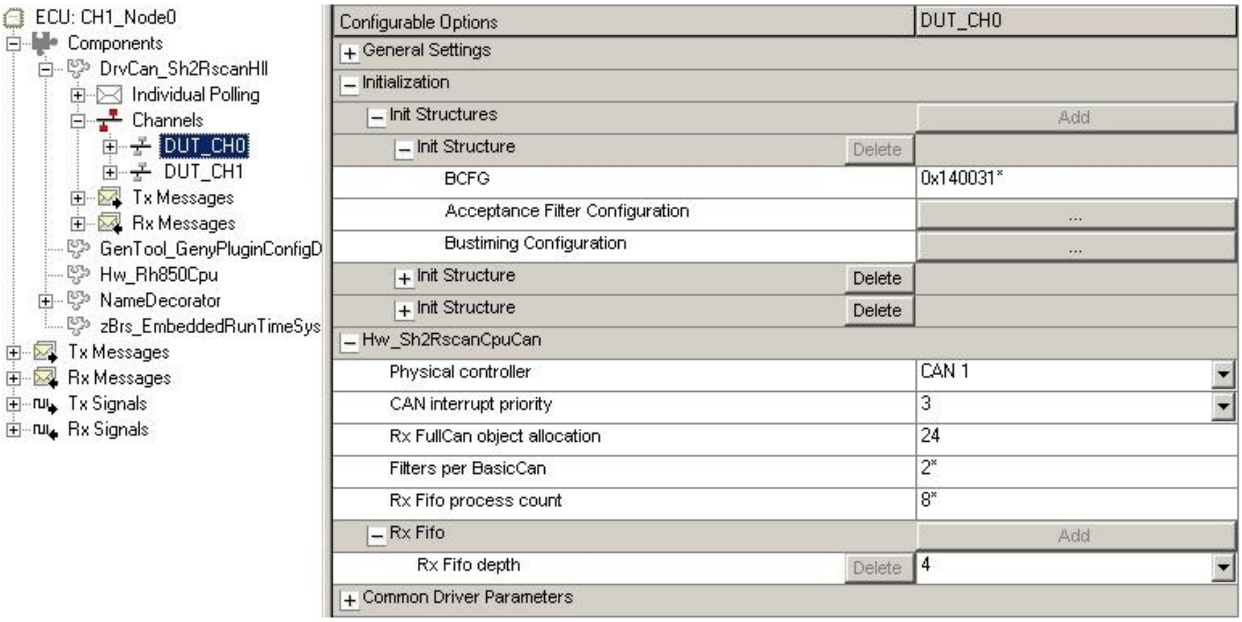


Figure 11-3 GENy Channel Specific Settings

Caution

The sum of the shared buffers used to allocate the receive objects over all channels must not exceed ("Maximum number of CAN channels" * 64)[7]. This includes the Rx FullCAN objects (1 buffer per actually assigned object; not the value of "Rx FullCAN object allocation) and the depth of all Rx BasicCAN objects (individually configurable amount of buffers, selected by the attribute "Rx Fifo depth") on all channels.

The sum of used acceptance filters over all channels must not exceed ("Maximum number of CAN channels" * 64)[7]. Each actually assigned Rx FullCAN object uses one filter and each Rx BasicCAN object uses the number of filters that is selected by the attribute "Filters per BasicCAN" on the corresponding channel.

The generation tool checks these and other restrictions (e.g. allowed selection for the attribute "Physical controller") to ensure valid configurations. Therefore it is mandatory to enter a valid value for the attribute "Maximum number of CAN channels"[6]. Refer to chapter 3 for further information.

[7] Refer to section 11.1.2 for the description of the attribute "Maximum number of CAN channels".

Attribute	Supported Values	Description
BCFG	register value	The value for the Channel Configuration Register.
Acceptance Filter Configuration	-	Opens the acceptance filter dialog, see section 11.1.3.1. If several init structures are created this is only possible for the first structure.
Bustiming Configuration	-	Opens the bustiming dialog to determine the value for BCFG, see section 11.1.3.2.
Physical controller	CAN 0 – CAN 7	Select the physical channel you want to assign to this logical channel. The value is enumerated the same way as referenced in the hardware manual. Depending on the selected derivative and configuration of the attribute “Maximum number of CAN channels” (see section 11.1.2) not all values may be available.
CAN interrupt priority	0 – 15	Select the interrupt priority level of this CAN channel’s interrupts that are configured if the driver has interrupt control. Depending on the selected derivative not all levels may be available. See section 7.2.3 and chapter 10 for further information.
Rx FullCAN object allocation	0 – nRXMBmax	You can configure as many receive FullCAN messages on this channel as specified here. This value is used to limit the selection for manual or automatic configuration - only the actually arranged FullCAN objects will be configured in hardware. The value of nRXMBmax equals “Maximum number of CAN channels” * 16.
Filters per BasicCAN	1 – 128	Select how many acceptance filters will be assigned to each Rx BasicCAN object on this channel; see section 11.1.3.1 for details. Depending on the selected derivative not all values may be available.
Rx Fifo process count	2 – 255	Select the maximum number of pending receive messages that are processed for each Rx BasicCAN object within one polling cycle respectively one interrupt occurrence. By adjusting this value it can be ensured that high FIFO loads will be evenly processed by the driver. Remaining messages are processed within the next polling cycle respectively the interrupt of the next received message on this channel. Select greater values if overruns occur.
Rx Fifo depth	4; 8; 16; 32; 48; 64; 128	Individually configure the depth (amount of assigned shared buffers) of every Rx FIFO, that is used as Rx BasicCAN object on this channel.

Table 11-3 GENy Channel Specific Settings

Note
As the RSCAN has restrictions regarding the receive buffers (e.g. no interrupt processing, no overrun detection) also consider configurations without Rx FullCAN objects. The large FIFO sizes and amount of filters that are possible for the BasicCAN objects give similar advantages as the usage of FullCAN objects. For many configurations this can be an alternative that above all is more effective regarding runtime and memory usage.

11.1.3.1 Acceptance Filter Configuration

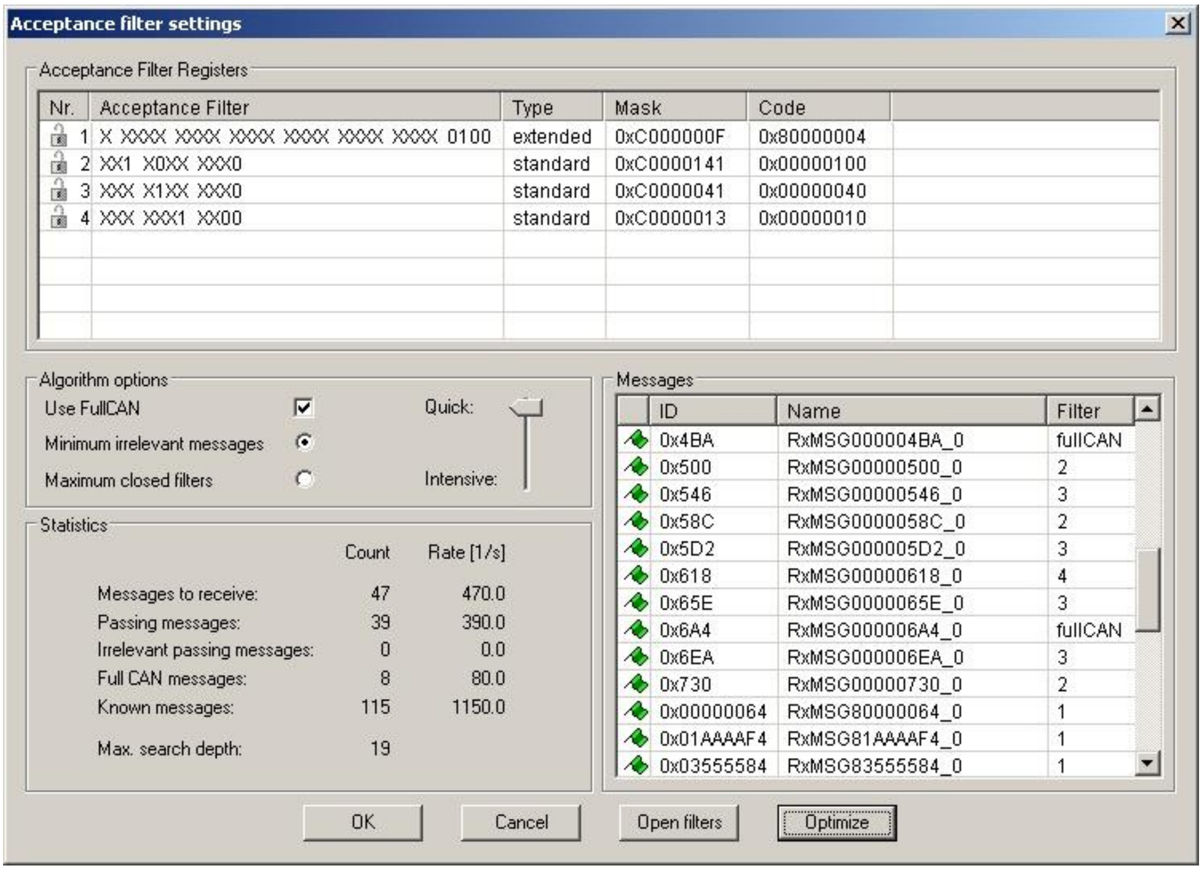


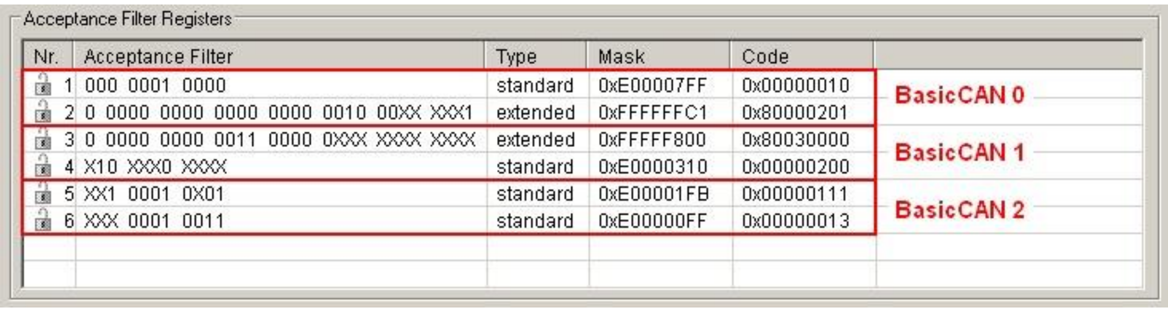
Figure 11-4 GENy Acceptance Filter Configuration

Attribute	Supported Values	Description
Acceptance Filter	representation of type, mask and code	The configured BasicCAN filters are shown. Each ID-bit is represented by "0/1/X", meaning must match "0", "1" or don't care "X". The number of filters can be adjusted by the attribute "Filters per BasicCAN" on the channel view.
Type	standard, extended	Select if the filter shall apply to standard or extended ID types. (Based on the database and configuration only one type may be available.)
Mask / Code	register values	The register values for this filter that will be configured in hardware.
Open filters	-	Open the filters completely to receive all messages.
Optimize	-	Configure the filters automatically to just receive IDs in the database if possible. A large number of filters allow better optimizations, but don't configure more filters than the optimization algorithm uses for message distribution. "Use FullCAN" tries to put as many messages as possible in FullCAN objects. Select the maximum number of available objects by adjusting the attribute "Rx FullCAN object allocation" on the channel view. This is useful when only a few number of BasicCAN filters are configured for example.

Table 11-4 GENy Acceptance Filter Configuration

11.1.3.1.1 Additional information if the feature “Multiple BasicCAN objects” is used

The dialog shows all BasicCAN acceptance filters for the respective channel. The amount of filters equals the product of configured BasicCAN objects and the number of filters per BasicCAN. An example configuration with 3 BasicCAN objects and 2 filters per object results in 6 filters as shown in figure 11-5. The first 2 filters (in accordance with the attribute “Filters per BasicCAN”) are assigned to the first BasicCAN object, the next 2 to the second BasicCAN object and so on.


Acceptance Filter Registers

Nr.	Acceptance Filter	Type	Mask	Code	
1	000 0001 0000	standard	0xE00007FF	0x00000010	BasicCAN 0
2	0 0000 0000 0000 0000 0010 00XX XXX1	extended	0xFFFFFFC1	0x80000201	
3	0 0000 0000 0011 0000 0XXX XXXX XXXX	extended	0xFFFFF800	0x80030000	BasicCAN 1
4	X10 XXX0 XXXX	standard	0xE0000310	0x00000200	
5	XX1 0001 0X01	standard	0xE00001FB	0x00000111	BasicCAN 2
6	XXX 0001 0011	standard	0xE00000FF	0x00000013	

Figure 11-5 GENy Acceptance Filter Assignment

Please note that a received message is stored in the first mailbox with a matching filter. After an identifier was compared against the FullCAN filters, it is compared against the BasicCAN filters in the order that is depicted in the dialog. This has to be considered when the feature “Multiple BasicCAN objects” is used. If filter number 1 in the example from figure 11-5 was open (all bits “don’t care”), all non FullCAN standard identifiers would be received by BasicCAN0 and BasicCAN2 would never receive any message.

Note

In some “Multiple BasicCAN” configurations it may be useful to assign certain BasicCAN messages to specific hardware objects as the FIFO depths or “Individual Polling” settings can be adapted to the actual communication aspects for example. As the optimization algorithms don’t consider this use case the filters have to be edited manually in this case.

Alternatively it is possible to configure and lock only several significant filters and then use the optimization functionality. But after doing so always check the result because manually configured filters may not always receive the pre-assigned identifiers as the message could match an automatically assigned filter that is compared first. Focus on filters with smaller numbers or add some “dummy filters” for earlier objects to achieve better results.

11.1.3.2 Bustiming Configuration

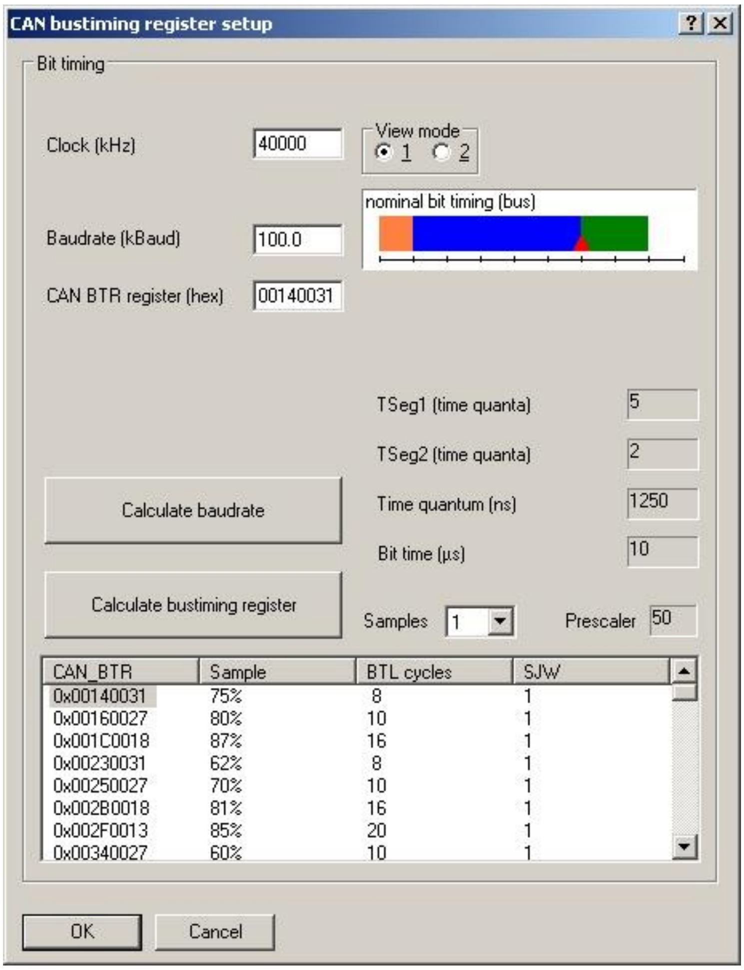


Figure 11-6 GENy Bustiming Configuration

Attribute	Supported Values	Description
Clock	CAN clock	Set the clock frequency that is provided to the CAN cell for baudrate generation (f_{CAN}). Consider the setting of the attribute “CAN external clock source” (see section 11.1.2).
Baudrate	baudrate	Set the baudrate to be used for this channel.
CAN BTR register	register value	Enter the value for the Channel Configuration Register (see attribute BCFG in section 11.1.3).
Calculate	-	Calculate possible Channel Configuration Register settings out of the entered baudrate or vice versa.
CAN_BTR, Sample, BTL cycles, SJW	-	Select specific channel configuration register values to adapt the sample point and sync phase to comply with your bus physics.

Table 11-5 GENy Bustiming Configuration

11.2 Manual Configuration

This section describes additional configuration options for special features that can only be configured via the user configuration file.

- Define `C_DISABLE_NESTED_INTERRUPTS` or `C_ENABLE_NESTED_INTERRUPTS` to control the nesting of the CAN interrupts. See section 10.1 for further information.
- Define `C_ENABLE_DIRECT_INTERRUPT_BRANCH` (and if needed additionally `C_ENABLE_ISRVOID`) to deactivate table reference as the method to handle CAN interrupts. See section 10.2 for further information.
- Define `C_ENABLE_OSEK_CAN_INTCTRL` to prohibit write accesses by the driver within the interrupt controller address space. See section 10.3 for further information.
- Define `C_ENABLE_EXTERNAL_WAKEUP_SUPPRESSION` to disable the external wakeup functionality. See chapter 4 for further information.
- See sections 7.2.6 to 7.2.9 for options that control different RAM test features.
- See section 10.2.1 for information on how to set up the interrupt vector table when using IAR compiler.

12 Known Issues / Limitations

1. Due to hardware limitations the feature CAN RAM check is not supported for receive mailboxes (no write access is possible for these objects).
2. Due to hardware limitations receive FullCANs cannot be processed in interrupt context and no overruns can be detected for these objects.
3. With default configuration the driver needs exclusive write access to all EI level interrupt control registers (ICn) that are related to a CAN interrupt source (see section 7.2.3 and chapter 10 for further information.).
4. Refer to chapter 4 for restrictions when using the sleep/wakeup functionality. Additionally the global stop mode of the RSCAN is not supported.
5. When using multiple initialization structures no multiple acceptance filter configurations are supported by the driver. The filter settings are always derived from the first structure. Use several structures only to arrange multiple baudrate configurations.
6. When using the feature Multiple ECU Configurations it is not supported to use a logical channel in more than one identity. The only exception is the first logical channel which can be present in any identity if it is also mapped to the physical channel CAN0. This limitation does not apply to the usage of physical channels: Every available physical channel can be used in any identity and the same physical channel can be used in as many identities as needed (if it is referenced by different logical channels).
7. For derivatives that incorporate multiple RSCAN units only the first one (RSCAN0) is supported by the driver.

For latest information about issues or limitations of the actually used derivative please contact the hardware manufacturer.

13 Contact

Visit our website for more information on

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

www.vector.com