Introduction and Support

This application note describes how to use CAN controllers of various Rensas microcontroller products.

Several different CAN controller types are available; so this application note is collecting frequently asked questions and hot topics for all of them. By using the index, the user may locate the answers in one or several chapters. Nevertheless, the content of this application note does not make any claim to be complete.

Due to this, the application note may be updated without further notice in shorter time intevals.
Proposals for improvement are always highly welcome.
For proposals and support, please contact <*device_support.micro-eu@lm.renesas.com*>.

Target Device

V850/Xx1, 78K0/Xx1 and earlier	FCAN / DCAN	CAN Controller types
V850/Xx2, V850/Xx3, 78K0/Xx2, 78K0(R)/Xx3:	AFCAN/DAFCAN	CAN Controller types
V850/Xx4:	FCN/DCN	CAN Controller types
SH:	RCAN...	CAN Controller types
RL78/X1x:	RS-CANLite	CAN Controller types
RH850/P1x-C:	M_(TT)CAN	CAN Controller types
RH850/X1x:	RS-CAN	CAN Controller types
RH850/X1x, E2x and later	RS-CANFD V2/V3	CAN Controller types
RH850/U2x and later	RS-CANFD V4	CAN Controller types

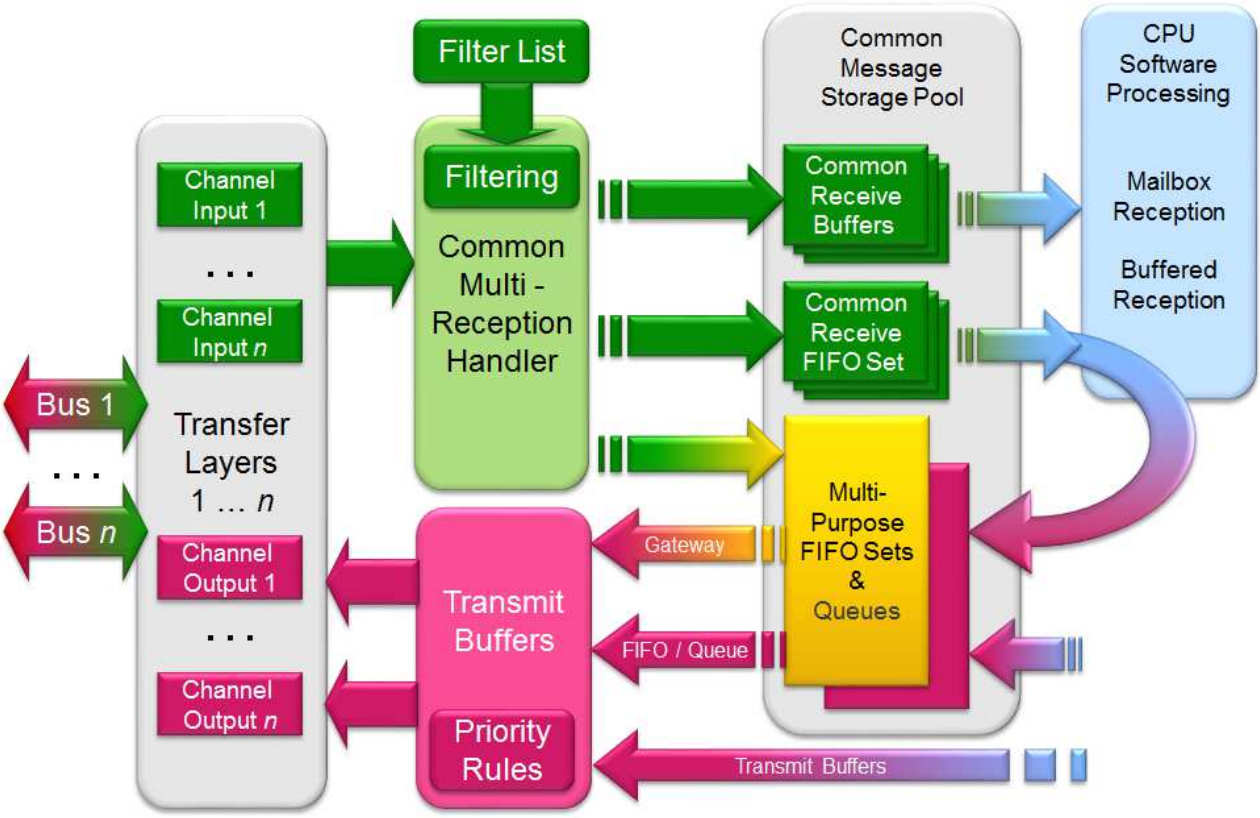

Figure 1.1 State of the art CAN Controller RS-CANFD V4

Note: Subsequent pages may be partly blank or have interleaved chapter numbering. This is by intention, as this application note is continuously improved.

Terminology Index

Issue Solving Proposal Index

1. CAN Controller Evolution in Renesas Microcontroller Products

1.1 Abstract

This chapter covers:
• Which migration paths regarding software can be performed with what amount of effort
• Which functionality must be considered new when migrating
• Software implementation and integration hints

The chapter assumes that the evolution steps of Renesas CAN controllers is understood as follows:
(older CAN controllers) --> AFCAN --> FCN --> RS-CAN --> RS-CANFD

M_(TT)CAN controllers are not Renesas IP, therefore there are no recommended migration paths between those and Renesas CAN controllers.

In addition, we assume that a migration with two evolution steps at a time is not possible without reconstructing the software drivers from scratch.

1.2 Older CAN Controllers

DCAN, FCAN and RCAN(...) are older CAN controllers. When migrating to a microcontroller state-of-the-art, a migration strategy cannot be recommended, apart from starting a new CAN controller software from scratch.
The reason for this is, that there are too many differences between older CAN controllers and newer ones, which are affecting the following:

• Hardware architecture and (new) functional features
• Implementation
  • Layout of registers (SFR)
  • Interrupt sources
  • Clock setting requirements
  • Algorithmic requirements for initialization, operation and error handling
  • Functional delays of state engines

In the following, we will show the migration aspects regarding these criteria for newer CAN controller types.

1.3 AFCAN CAN Controllers

With AFCAN, many new features have been introduced, so that all categories of migration would require to be adapted. For this reason, a migration path from older CAN controllers to AFCAN does not exist and CAN driver software must be rewritten almost completely.

1.3.1 AFCAN Hardware Architecture and New Functional Features

The hardware architecture is changed from multiple channel unit with shared memory into a single-channel module. Thus, programmable automated processing of messages among channels by hardware is no longer possible. At the same time however, the diagnosis channel functionality is introduced (“DAFCAN”). This concept supports (filtered) hardware message routing from several channels into one.
As an advantage, the new architecture is more compact and has lower power consumption.

1.3.2 AFCAN Implementation Changes to Older CAN Controllers

Table 1.1 AFCAN Implementation Changes to Older CAN Controllers

Category	Item	Description
Register Layout	Channel Message Box Debug Information	One channel per CAN Controller. Layout of Message Box contents optimized. Reduced to TEC, REC and Bus Off
Interrupt Sources	all	Edge triggered, pending flag needs not to be cleared any more
Clocking	Protocol requirement	8 MHz for 1 Mbit/s required

Table 1.1 AFCAN Implementation Changes to Older CAN Controllers

Category	Item	Description
Algorithms / Functions	Message Searching	Replaced by RX/TX History functionality
	Data Content Filtering	Removed, filtering restricted on ID of messages only
	Time Trigger System	Using additional timer resource, SOF and EOF triggers
	Operation Modes	State machine for operation mode transition introduced Power Save Modes introduced (Sleep, Stop)
	New: Block Transfers	ABT Mode (up to 8 messages sending with single trigger)
	New: RX/TX History	Correlate received and sent messages with boxes in sequence
	New: Diagnosis Modes	RX-Only, Single Shot, Self Test Loop
	New: MUC Flag	Indicates busy phases of message buffers, to be considered by software when reading messages during reception
Delays	Prioritized CPU access	Latency of CPU access (wait states) reduced
	Operation Modes	Changes of Operation Modes must be confirmed by reading back the operation mode status
	Bus Off Recovery	Can now alternatively be handled by software

1.4 FCN CAN Controllers

The introduction of FCN CAN controllers was mostly done in order to get a performance boost. While basically the architecture of AFCAN was kept, the whole register map and the internal peripheral bus access was heavily improved. At the same time, more message boxes and longer history lists were introduced, so that a much larger associated internal RAM was implemented.

1.4.1 FCN Hardware Architecture and New Functional Features

Besides the performance mentioned above, some new features were introduced:

- Centralized “New Data Flag” register, which groups all “New Data” flags from all message buffers:
  This allows easy lookup of new receptions without scanning the whole RX history list.
- Remote frames can now also be received in RX buffers with masking option, not only in FullCAN TX buffers:
  Flexibility when receiving remote frames is heavily improved by that. Remote frames get the same possibility of use like data frames.
- History list recording of TX and RX messages can be disabled per message buffer:
  The feature avoids overflow of history lists, if many message boxes are not handled by the history list scanning process (for example, they are polled).

1.4.2 FCN Implementation Changes to AFCAN CAN Controllers

Table 1.2 FCN Implementation Changes to AFCAN CAN Controllers

Category	Item	Description
Register Layout	All	New register layout due to optimized peripheral bus access. Up to 128 message boxes per channel.
Interrupt Sources	Transmission Abortion	Additional interrupt source, shared with Wake-Up interrupt
	ECC Check of RAM	Additional interrupt source, depends on implementation
Clocking	Protocol requirement	40 MHz for 1 Mbit/s required, due to additional sampling stage
Algorithms / Functions	Block Transfers	Up to 32 messages per block, depending on implementation
	RX/TX History	Up to 96 record entries (RX), 32 record entries (TX), depending on implementation
	Hardware Filter Masks	Doubled amount to 8 masks per channel
	New: New Data Register	Grouping of all DN flags of all message buffers
	New: History Disabling	Recording in history lists can be disabled per message buffer
	New: Remote Frames	Reception in RX buffers possible
Delays	Faster Peripheral Bus	Latency of CPU access (wait states) reduced
	RAM Initialization, Soft Reset	Self-Initialization of local CAN RAM: software must wait for that after each hard- or soft-reset

1.5 RS-CAN CAN Controllers

With the introduction of the RS-CAN CAN controller, the general approach for handling CAN was revised. The most important topic here is the migration from a message *box* approach to a message *stream* approach.
At the same time, the common resource (RAM, buffer) sharing concept among several channels was reinvoked.
Faster technology with higher clock rates and higher gate density (higher complexity of design) had allowed these steps previously, while increasing data traffic on the CAN bus at higher data rates required a new concept, too.

1.5.1 RS-CAN Hardware Architecture and New Functional Features

In the following, the most important architectural differences to the AFCAN and FCN CAN controllers are highlighted. Due to these differences, there is no simple migration path that could be recommended in general. Even more, in order to get best advantages of the new architecture and features, a new start from scratch for software drivers is making sense.

New features / functionality:
- Multi-channel architecture with shared RAM storage, flexible usable of RAM resources
- Fixed amount of hardware filters replaced by filter lists for each channel
- Reception process allows several (up to 8) targets, hereby copying a message for each target
- Reception targets are FIFO structures and classical message boxes
- Classical receive message boxes do not support interrupts
- Transmission can be performed from either/both FIFO structures, prioritized queues and classical transmit boxes
- Automated routing of complete messages between receive and transmit paths among all associated channels, not just a single diagnosis channel, but using all channels
- Time stamping now fully integrated, triggered on either SOF or EOF
- Logical handles (labels) for receive and transmit messages
- Data length (DLC) supervision on reception
- Internal self test bus linking several channels

Removed features / functionality:
- Receive History List
- Receive interrupts of classical message boxes
- Overwriting protection of classical message boxes
- Transmit block transfer replaced by transmit FIFO structures
- Multi-receive block (MRB) functionality replaced by receive FIFO structures
- Wake up interrupt: functionality moved from CAN controller to port/microcontroller top level

1.5.2 RS-CAN Implementation Changes to FCN CAN Controllers

Due to the new architecture, neither register tables nor algorithms for reception and transmission can be compared between RS-CAN and the previous CAN controller types.

1.5.3 Memory Layout of RS-CAN CAN Controllers

In order to define a correct configuration, it is inevitable to understand the layout of the associated CAN controller memory in detail. In the following diagram, the memory layout of the 6-channel, 3-channel RS-CAN and dual channel and single channel RS-CANLite CAN controllers is shown.

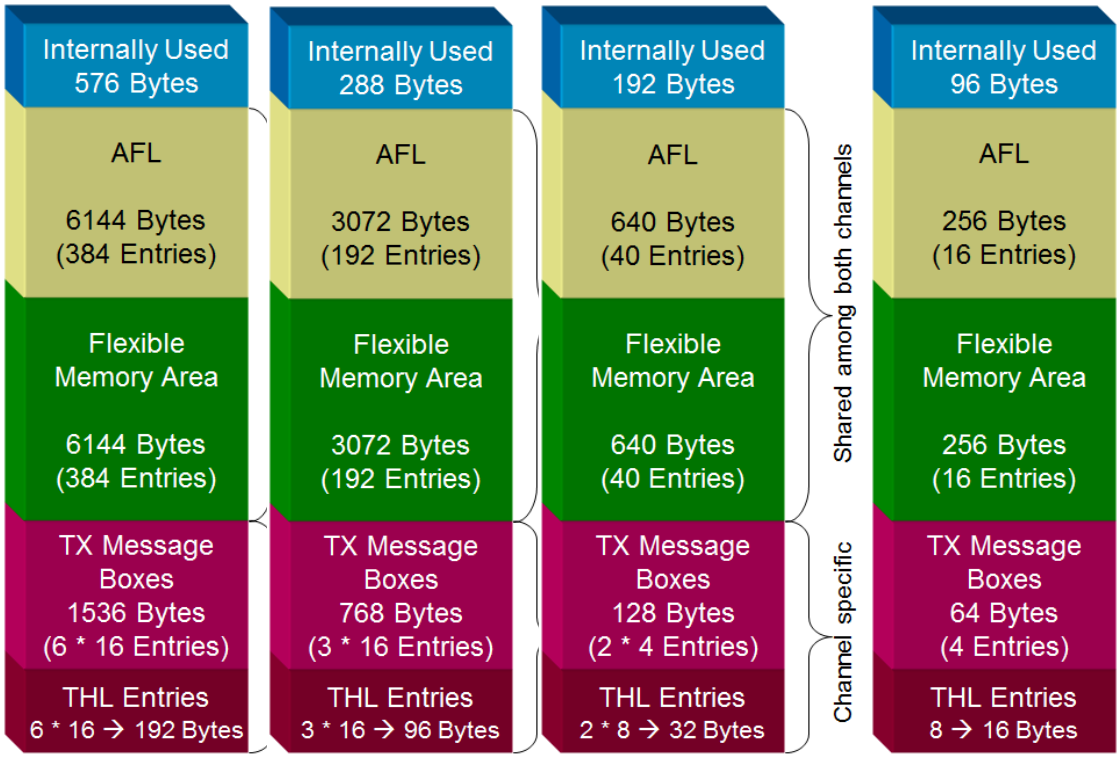

Figure 1.1 Memory Layout of RS-CAN: 6-channel, 3-channel; RS-CANLite: 2-channel, 1-channel

The channel specific memory parts do not require dedicated consideration, because these resources are not shared and simply stay unused, if not allocated.
However, the shared memory parts require explicit partitioning, which are the acceptance filtering lists (AFL) and the common reception resources (flexible memory area). The following rules must be obeyed:

- AFL memory parts cannot be allocated to reception resources and vice versa.
- AFL memory parts are shared among all channels. Each rule takes one entry. It is not allowed to define more rules than the maximum allowed total amount (i.e., 384 on a 6-channel RS-CAN).
- If below the limit, the amount of AFL rules can be individually assigned to channels.

- The flexible memory area is shared for all reception resources, such as standard boxes and FIFO elements.
- A standard reception box is occupying one entry.
- A FIFO (either RX only or TX/RX [“multi-purpose”]) consumes one entry for each of its elements. Thus a FIFO with a depth of 128 will consume 128 entries.
- The maximum number of entries must not be exceeded by adding up all boxes and FIFO elements.

When using the RAM testing functionality of RS-CAN, the RAM area to be testable is the sum of all usable memory cells shown above plus an internally used storage amount of 96 bytes per channel; i.e., 14592 bytes for the 6-channel version.
RAM testing is organized in pages with 256 bytes each, using 64 longword-based registers for one page.
For the 6-channel version of RS-CAN, thus there are 57 complete pages.
For the 3-channel version of RS-CAN there are 7296 bytes, thus 28 pages and one page with 128 bytes; in total 28.5 pages.
Even though the physical CAN memory might be larger, a testing access to these sections is prohibited.

1.6 RS-CANFD CAN Controllers

Within the introduction of CAN-FD, the RS-CANFD Controllers were developed from the RS-CAN Controllers, and are still continuously improved.
The major version of the RS-CANFD Controller can be asked from the Renesas support (e-mail address see front page).

- RS-CANFD V2: CAN-FD ISO 11898-1:2015 compatible. RS-CANFD V2 provides a RS-CAN compatibility mode. In compatibility mode however, CAN-FD functionality is disabled.
- RS-CANFD V3: Enhanced version of RS-CANFD, which no longer supports compatibility to RS-CAN, but has advantages of new CAN-FD features and larger memory, simplifying its usage.
- RS-CANFD V4: Enhanced version of RS-CANFD, with again larger memory, higher data rates and additional essential features regarding hardware gateway, buffer and transceiver sharing.

1.6.1 Implementation Differences of RS-CANFD Controllers

Table 1.3 RS-CANFD V2 Implementation Differences in RS-CAN Compatibility Mode

Category	RS-CAN	RS-CANFD V2 (RS-CAN Mode)
Classical CAN Protocol	ISO 11898-1:2003 compatible	
Register Layout	Reference	No changes
Interrupt Sources		
Clocking		
Algorithms / Functions		CAN Controller RAM Size is increased: More RAM Pages available (RAM Test Functionality) CAN Controller RAM ECC verification at transmission and on CPU reading
Delays		CAN Controller RAM Initialization after Reset: May take longer time Operation Mode changes: slightly different execution time due to changed state engines Transmission Latency (from software trigger to transmission start on CAN bus): slightly different due to modified protocol engine

Table 1.4 RS-CANFD V3 Implementation Differences in RS-CANFD Mode of V2

Category	RS-CANFD V2 (RS-CANFD mode)	RS-CANFD V3
CAN-FD Protocol	CAN-FD ISO 11898-1:2015 compatible Bit timing generation according to CAN-FD	
Register Layout	New register memory addressing layout due to larger structures Expanded Message Object/Buffer layout Merging of Transmit Buffers	New register memory addressing layout due to larger structures Expanded Message Object/Buffer layout No merging Transmit Buffers (enlarged, thus no longer required)
Interrupt Sources	No changes	
Clocking	No changes	CAN-FD operation: max. 80 MHz peripheral clock and 40 MHz communication clock
Algorithms / Functions	CAN-FD Feature Set: FDF, BRS, ESI, TDC, Frames with 64 Bytes, Data Padding CRC field generation / check Timestamp also at FDF->res edge Restricted Operation Mode FD Tolerant Operation Mode DMA Support functionality	New CAN-FD Features: FD-Only Mode, Classical-Only Operation Modes Doubled amount of AFL rules Doubled amount of TX buffers Message Labels increased to 16-bit All buffers now can cover 64 bytes of payload data (TX and RX)
Delays	Default	CAN Controller RAM Initialization after Reset: May take longer time

Table 1.5 RS-CANFD V4 Implementation Differences to V3

Category	RS-CANFD V3	RS-CANFD V4
CAN-FD Protocol	CAN-FD ISO 11898-1:2015 compatible	CAN-FD ISO 11898-1:2015 compatible Fulfills CiA 601.x recommendations
Register Layout	-	New register memory addressing layout due to higher bit resolution and functional enhancements Expanded Message Object/Buffer layout
Interrupt Sources	-	Additional interrupts for FIFOs and Queues
Clocking	Max. 80 MHz peripheral clock and 40 MHz communication clock	CAN-FD operation: max. 80 MHz peripheral clock, 80 MHz communication clock and 160 MHz memory access clock
Algorithms / Functions	CAN-FD Feature Set: FDF, BRS, ESI, TDC, Frames with 64 Bytes, Data Padding CRC field generation / check Timestamp at FDF->res edge or SOF Restricted Operation Mode FD Tolerant Operation Mode FD-Only Mode Classical-Only Operation Mode DMA Support functionality	CAN-FD Feature Set: See V3 Improvements: Amount of AFL rules increased by 50% Doubled amount of TX buffers 4 TX Queues per channel Flexible channel interconnection Flexible TX buffer sharing by two channels HW-Gateway routing to TX Queues Overwrite / Replace mode of TX Queues Overwrite / Discard mode of FIFOs FIFO stopping and One-Frame signaling Transmit History in Gateway Mode Soft-Reset function
Delays	Default	CAN Controller RAM Initialization after Reset: May take longer time

Table 1.6 RS-CANFD Changed Property Comparison

Feature	RS-CANFD V2	RS-CANFD V3	RS-CANFD V4
AFL Rules total / max. per channel	64 * Channels / 127	128 * Channels / 255	192 * Channels / 384
Receive / FIFO Buffers	23~89 Objects * Channels	47~179 Objects * Channels	256~972 Objects * Channels
Transmit Buffers	16 * Channels	32 * Channels	64 * Channels
Transmit Queues	1 * Channels	1 * Channels	4 * Channels
Max. Payload Size for Transmit Buffers	20 Bytes, 64 Bytes by merging three buffers	64 Bytes for all buffers	64 Bytes for all buffers
Max. Payload Size for Standard Receive Buffers	20 Bytes	64 Bytes	64 Bytes
Label (Pointer) Size for Transmission	8 Bit	16 Bit	16 Bit
Label (Pointer) Size for Reception	12 Bit	16 Bit	16 Bit
Operation Modes	RS-CAN Compatible Mode CAN-FD Operation Mode CAN-FD Tolerant Mode - - Restricted Operation Mode External Self-Test Mode Internal Self-Test Mode Receive-Only Mode -	- CAN-FD Operation Mode CAN-FD Tolerant Mode CAN-FD Only Mode Classical CAN Only Mode Restricted Operation Mode External Self-Test Mode Internal Self-Test Mode Receive-Only Mode -	- CAN-FD Operation Mode CAN-FD Tolerant Mode CAN-FD Only Mode Classical CAN Only Mode Restricted Operation Mode External Self-Test Mode Internal Self-Test Mode Receive-Only Mode Flexible CAN Mode

1.6.2 Memory Layout of RS-CANFD CAN Controllers

Similarly to RS-CAN and RS-CANLite, in RS-CANFD the memory organization is a very important aspect when configuring the CAN controller.
Due to the high variations in object sizes of CAN-FD, the memory organization no longer can be given in receive or transmit object counts, but in absolute amounts of bytes of memory, which are available and which are required to store a receive or transmit object.
For the *n*-channel types of RSCAN-FD V2, V3 and V4, the memory layout is given as shown below:

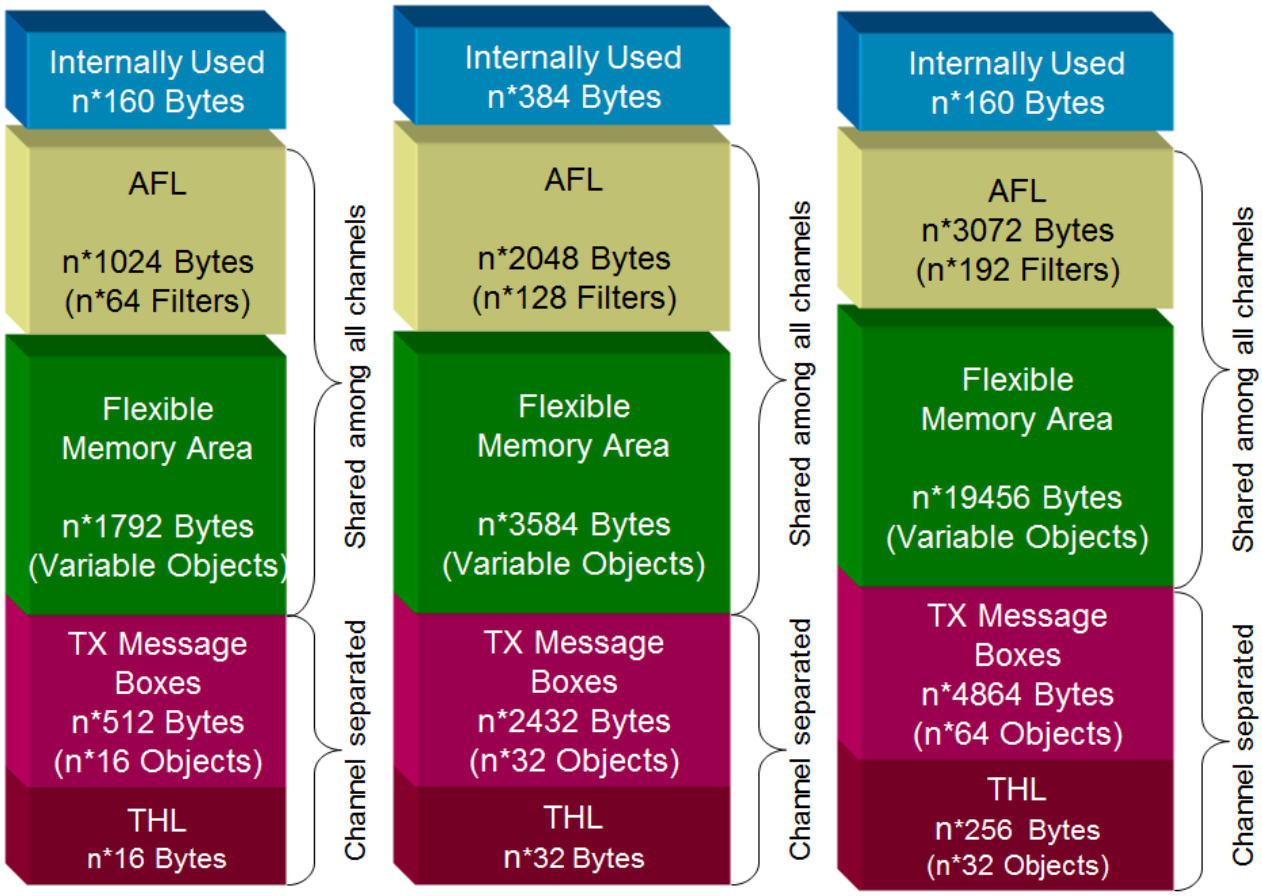

Figure 1.2 Memory Layout of RS-CANFD V2, RS-CANFD V3, RS-CANFD V4: *n*-channel versions

When using the RS-CAN compatibility mode of RS-CANFD V2, the memory layout changes to the RS-CAN layout. See *1.5.3 Memory Layout of RS-CAN CAN Controllers* for details.
The channel specific memory parts do not require dedicated consideration, because these resources are not shared and simply stay unused, if not allocated.
However, the shared memory parts require explicit partitioning, which are the acceptance filtering lists (AFL) and the common reception resources (flexible memory area). The following rules must be obeyed:

- AFL memory parts cannot be allocated to reception resources and vice versa.
- AFL memory parts are shared among all channels. Each rule takes one entry. It is not allowed to define more rules than the maximum allowed total amount (i.e., 384 on a 6-channel RS-CANFD V2).
- If below the limit, the amount of AFL rules can be individually assigned to channels.

- The flexible memory area is shared for all reception resources, such as standard boxes and FIFO elements.
- A standard reception box is occupying 12+*d* bytes, where *d* is the message size of CAN-FD in bytes.
  The message size *d* can be adjusted to be 8, 12, 16 or 20 bytes (RS-CANFD V2).
  The message size *d* can be adjusted to be 8, 12, 16, 20, 24, 32, 48 or 64 bytes (RS-CANFD V3, V4).
- A FIFO (either RX only or TX/RX [“multi-purpose”]) consumes 12+*d* bytes for each of its elements.
  The message size *d* can be adjusted to be 8, 12, 16, 20, 24, 32, 48 or 64 bytes.

Thus a FIFO with a depth of 128 and payload size of 64 bytes will consume 9728 bytes. It cannot be activated on a 3-channel RS-CANFD V2 unit, because it exceeds the limit of 5376 bytes.
The maximum number of entries must not be exceeded by adding up all boxes and FIFO elements.
For RAM testing, the approach is the same as for RS-CAN(Lite). See *1.5.3 Memory Layout of RS-CAN CAN Controllers* for details.

1.6.3 Merging Transmit Buffers in RS-CANFD V2

When using a transmit ("multi-purpose") FIFO for transmission, it can be assigned to any transmit buffer, and the size of the transmit buffer needs not to be considered.

However, when using classical transmit buffers, for RS-CANFD V2 transmit buffers need to be merged, depending on the payload size to be transmitted. For RS-CANFD V3, this is no longer required, because all transmit buffers are supporting the full 64-byte payload capacity.

A transmit buffer in RS-CANFD V2 can store up to 20 payload bytes. If more bytes shall be stored, merging has to be activated by setting the *Merge Mode* flag in the global configuration. In *Merge Mode*, it is allowed to cross the boundaries of the transmit message buffers with additional payload data, so that the one or two next buffers would be unusable as separate buffers, just being a storage for the additional payload.

Merging of transmit buffers can be done for the lowest 6 buffers only, and the base of merged buffers must be either buffer 0 or buffer 3. Either one, two or three buffers can be grouped in *Merge Mode*, without setting any additional flags; just by specifying the payload size. Merging two buffers yields a maximum payload size of 48 bytes, merging three buffers yields 64 bytes payload size.

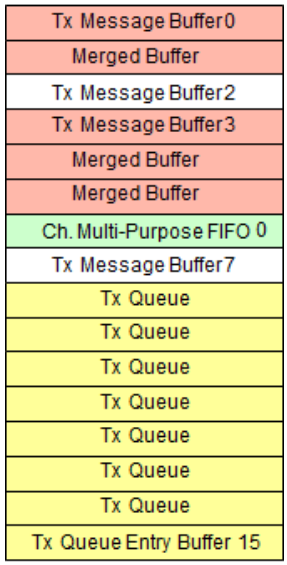

Figure 1.3 Merging Transmit Boxes Example for RS-CANFD V2

In the given example, buffer 0 is merged with buffer 1, to be a transmit message box for CAN-FD messages with up to 48 bytes payload. Buffer 2 is available for up to 20 bytes payload. With another merging, the buffers 3, 4 and 5 are grouped to fit for a 64 bytes payload.
Buffer 6 is assigned to a multi-purpose TX FIFO. It can transmit any payload size up to 64 bytes even though there is no merging for this buffer. The TX FIFO will manage the re-use of the buffer to sequentially transmit the payload without need of additional storage. The payload itself resides in the FIFO, where the memory is acquired.
Buffer 7 is again a buffer for a 20 bytes payload.

Buffers 8 to 15 are joined into a transmit queue. Transmit queues in RS-CANFD V2 are not supporting larger payloads than 20 bytes. Thus, the queue is set up for CAN-FD frames for up to 20 bytes payload size with a queue depth of 8 buffers.

Prioritized transmission is supported for this queue, while the FIFO assigned to buffer 6 will not support prioritized transmission among its content. Regarding transmission priority, all queue members, all buffers or merged buffers, and the front message of the FIFO will arbitrate for highest priority.

1.7 M_(TT)CAN Controllers

Within the evolution of CAN-FD, similarly like the RS-CANFD controllers, the M_(TT)CAN controllers were updated in several steps. Within Renesas products, the versions V3.0.1 and V3.2.1 are implemented. The following table shows the adapted topics which need to be considered when migrating.

Table 1.7 M_(TT)CAN Implementation Differences between Versions V3.0.1 and V3.2.1

Category	M_(TT)CAN V3.0.1	M_(TT)CAN V3.2.1
CAN Protocol	ISO 11898-1:2003 (classical CAN) CAN-FD protocol not in line with ISO	ISO 11898-1:2015 compatible (CAN-FD)
Register Layout	FBTP TEST CCCR BTP PSR TDCR IR, IE and ILS RX Buffer and FIFO Elements TX Buffer Elements	renamed to DBTP increased configuration range transmitter delay compensation moved to TDCR transmitter delay compensation moved to PSR EFBI and PXHD settings added CMR flag removed CME replaced by BRSE and FDOE renamed to NBTP reduced configuration range for prescaler increased configuration range for bit timing transmitter delay compensation moved from TEST PXE flag added for protocol exception event REDL and FLEC flags renamed new added with transmitter delay compensation offset and filter window length flags STE..., FOE..., ACKE..., BE..., CRCE... replaced by ARA..., PED... and PEA... flag EDL renamed to FDF according to ISO ESI and frame format selectable per transmission
Interrupt Sources	See changes of IR, IE and ILS registers	
Clocking	Peripheral (“host”) clock issue: At least 26,7 MHz required to operate at 1 Mbit/s classical CAN	Peripheral clock issue fixed; requirement to be faster than the communication clock. Higher resolution for bit timing
Algorithms / Functions		Transmitter delay compensation adapted Selection of ESI state per transmission possible Selection of classic or FD format per transmission
Delays	Due to CCCR.CCE issue workaround, the entry in INIT mode is delayed.	Regular delay when entering INIT mode (worst case the length of a frame).

2. Bus Transceivers and CAN Controller Operation Modes

2.1 Abstract

This chapter covers:

- Which operation modes are available in transceivers and CAN controllers
- Which operation modes can be combined between transceivers and CAN controllers
- Which consequences may happen if bad combinations of operation modes between transceivers and CAN controllers are used
- Is it possible to create a CAN bus without transceivers?
- What happens if a CAN transceiver gets disconnected or shortcuts on its CAN controller interface signals?

2.2 Overview

In the following table, it is shown which modes of CAN transceivers can be combined with modes of CAN controllers: Green: YES, can be combined, Red: NO, cannot be combined; do never combine, ***not even during state transitions***, Yellow: N/R, not recommended, but may be an intermediate state during state transitions.

Table 2.1 CAN Transceiver and Controller Mode Combinations

CAN Controller Modes	CAN Transceiver Modes						Remarks
	OFF	SLEEP	STANDBY	OFFLINE	RX-ONLY	ACTIVE	
RESET	YES	N/R	N/R	N/R	N/R	N/R	RESET tolerates all states.
STOP	N/R	YES	YES	YES	N/R	N/R	Transceiver should be inactive. Consider fast wake up and activity.
SLEEP	N/R	YES	YES	YES	N/R	N/R	
HALT / INIT	N/R	YES	YES	YES	YES	YES	Avoid floating port signals.
RX-ONLY	NO	N/R	N/R	N/R	YES	YES	Allow bus activity detection at least.
OPERATING	NO	NO	NO	NO	NO	YES	Do not block transmissions.
SELF TEST EXT.	NO	NO	NO	NO	NO	YES	Self-reception in operating mode.
SELF TEST INT.	YES	YES	YES	YES	YES	YES	Works independent of transceiver.

The modes are available in the various CAN controllers by different settings. See the following table and refer to the user's manual of your product, in order to find out how to set the operation mode.

Table 2.2 CAN Controller Types and Operation Modes

CAN Controller Modes	CAN Controller Types (*n*: Channel Number)						
	FCAN / DCAN	AFCAN / DAFCAN	FCN / DCN (*x*: F or D)	RCAN	RS-CAN, RS-CANLite	RS-CANFD V2, V3	M_(TT)CAN
					Set Global Operation / Clear Global Stop first.		
RESET	Clear CGST.GOM	Clear CnGMCTRL. GOM	Clear xCNnGMCLCTL. xCNnGMCLPWOM	Set CCTLR. CANM0	Set CnCTR. CHMDC = 1	Set CFDCnCTR. CHMDC = 1	(not available)
STOP	Set CnCTRL.STOP	Set CnCTRL. PSMODE1	Set xCNnCMCLCTL. xCNnCMCLMDPF1	Set CCTLR. SLPM	Set CnCTR. CSLPR	Set CFDCnCTR. CSLPR	Set CCCR.CSR
SLEEP	Set CnCTRL.SLEEP	Set CnCTRL. PSMODE0	Set xCNnCMCLCTL. xCNnCMCLMDPF0	Not available			
HALT / INIT	Set CnCTRL.INIT	Set CnCTRL. OPMODE = 0	Set xCNnCMCLCTL. xCNnCMCLMDOF = 0	Set CCTLR. CANM1	Set CnCTR. CHMDC = 2	Set CFDCnCTR. CHMDC = 2	Set CCCR.INIT
RX-ONLY	Set CnDEF.MOM	Set CnCTRL. OPMODE = 3	Set xCNnCMCLCTL. xCNnCMCLMDOF = 3	Set CTCR = 3	Set CnCTR.CTME CnCTR.CTMS = 1	Set CFDCnCTR.CTME CFDCnCTR.CTMS = 1	Set CCCR.MON
OPERATING	Clear CnCTRL.INIT	Set CnCTRL. OPMODE = 1	Set xCNnCMCLCTL. xCNnCMCLMDOF = 1	Clear CCTLR.SLPM CCTLR.CANM	Clear CnCTR.CHMDC CnCTR.CSLPR	Clear CFDCnCTR.CHMDC CFDCnCTR.CSLPR	Set CCCR.CMR Clear CCCR.INIT
SELF TEST INT.	Set CTBR = 5 (2 channels needed at least)	Set CnCTRL. OPMODE = 5	Set xCNnCMCLCTL. xCNnCMCLMDOF = 5	Set CTCR = 7	Set CnCTR.CTME CnCTR.CTMS = 3	Set CFDCnCTR.CTME CFDCnCTR.CTMS = 3	Set CCCR.MON Set TEST.LBCK

Table 2.2 CAN Controller Types and Operation Modes

CAN Controller Modes	CAN Controller Types (*n*: Channel Number)						
	FCAN / DCAN	AFCAN / DAFCAN	FCN / DCN (*x*: F or D)	RCAN	RS-CAN, RS-CANLite	RS-CANFD V2, V3	M_(TT)CAN
					Set Global Operation / Clear Global Stop first.		
SELF TEST EXT.	Not available			Set CTCR = 5	Set CnCTR.CTME CnCTR.CTMS = 2	Set CFDCnCTR.CTME CFDCnCTR.CTMS = 2	Set TEST.LBCK

2.3 CAN Bus without Transceivers

Within internal point-to-point connections of nodes, the CAN bus can be used without transceivers. However, some external circuitry is required at least, and the solution is not in line with the ISO specification. On the other hand, as a CAN transceiver also limits the throughput and bandwidth, faster communication speed can be achieved than specified within ISO 11898-2.

Note: Using faster communication speeds may be outside of the specification of a device, and thus may not be guaranteed under all conditions.

In the following figure, the wiring of two devices using a CAN channel is shown.

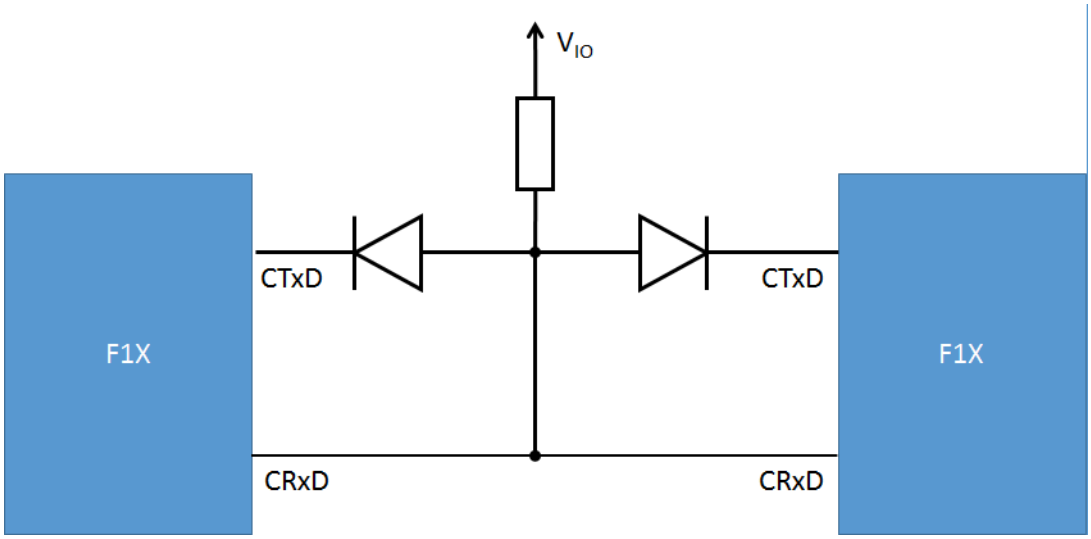

Figure 2.1 CAN Bus without Transceivers

The value of the resistor depends on the bus speed and the driving capabilities of the connected device ports. For the diodes, schottky types are recommended (higher speed and lower threshold voltage).

2.4 CAN Transceiver / Controller Interface Distortions

The reaction on distortions on the interface signals between CAN transceiver and CAN controller are mostly identical on all Renesas CAN controller types. Exceptions are shown below.
We are distinguishing between interruptions (a signal is no longer connected) and shortcuts (a signal is fixed to a level). Further, it is required to distinguish between the signals themselves, i.e., RX and TX. Any other signals of the CAN transceiver are not covered by this chapter, because Renesas controllers are not supporting them natively (means: by hardware on its own).
The following table shows the details. This shall serve for finding the distortions by their effects.

Table 2.3 CAN Transceiver / Controller Interface Signal Error Scenarios

Signal / Scenario	TX (“CTXD*n*”)	RX (“CRXD*n*”)
Open / unconnected	The transceiver is pulling up the TX signal to recessive (high) level. Transmission is not possible, the CAN bus remains unaffected. Reception is not possible, because the CAN controller cannot acknowledge. Any transmission attempt in the CAN controller will cause TEC errors, resulting in “Bus-Off” condition after 390 bit times. Detail: (a) Any reception attempt of a frame on the bus will cause an increase of the REC error counter; after 16 frames the CAN controller will be in error passive state. Detail: (b)	As RX is an input, see the rows of shortcut scenarios for applicable effects.
Shortcut to ‘1’ level (recessive)	See “open / unconnected” of TX.	Transmission is not possible, but error frames will be seen on the CAN bus, until the CAN controller reaches error passive state as indicated in detail (a). Reception is not possible, no frames are recognized.
Shortcut to ‘0’ level (dominant)	The transceiver is blocking the dominant level after a maximum distortion time according to its specification. During the distortion time before cut-off by the transceiver, the CAN bus is blocked (“locked dominant”). Any transmission attempt in the CAN controller will cause TEC errors, resulting in “Bus-Off” condition after 390 bit times. Any reception attempt of a frame on the bus will cause an increase of the REC error counter; after 16 frames the CAN controller will be in error passive state. Detail: (b)	The CAN controller recognizes a “locked dominant” condition. Integration and transmission on the CAN bus is not possible. Depending on the CAN controller implementation, a fault indication (interrupt, error) or a locked condition occurs. (c)
Shortcut between TX and RX	If the driver strength of CAN controller TX dominates, transmissions will be possible but endlessly repeated, because no acknowledge can be received. After 16 transmission attempts, the CAN controller will be error passive. Reception will not be possible. If the driver strength of CAN transceiver RX dominates, transmissions will not be possible (see detail (a). Receptions are possible, with conditions as shown in detail (b).	

a. Assuming that the CAN controller did have a TEC=0, when the scenario starts. 17 transmission attempts with 3 active error frames and 14 passive error frames with associated spacing will occur, until the bus-off state is reached at a value TEC=256.
b. Assuming that the CAN controller did have a REC=0, when the scenario starts.
If there are other stations on the CAN bus, who are providing acknowledge, then reception is possible.
c. FCAN, DCAN, RCAN, FCN, DCN, M_(TT)CAN: Locked condition. Soft reset of CAN controller is required.
RS-CAN, RS-CANFD: Fault indication. Can recover by operation mode setting.

3. CAN Bus Errors and Recovery of CAN Controller

3.1 Abstract

This chapter covers:
• What are the roles “transmitter” and “receiver” of a CAN controller according to ISO
• What happens, if a CAN transmitter does not get acknowledge from any receiver
• What happens, if a CAN transmitter does not see its transmitted bit value on the CAN bus
• What happens, if a CAN transmitter does not see its transmitted error frame on the CAN bus
• What happens, if a CAN receiver cannot receive its own acknowledge flag
• What happens, if a CAN receiver sees any error on the CAN bus (message format or else)
• What happens, if the CAN bus is permanently dominant
• What happens, if operation mode changes are performed under error conditions
• What happens, if bus off recovery must be performed

3.2 Roles of CAN controllers

According to ISO 11898-1, a CAN controller can be either “Transmitter” or “Receiver”.

• A “Transmitter” is a CAN controller, which arbitrates or is currently sending a message, after it has won the arbitration.
• A “Receiver” is a CAN controller, which is not currently sending a message, or which has lost the arbitration.

If a CAN controllers detects errors, it is increasing its error counters. Any successful transmission or reception decreases the error counters. Depending on the level of the error counters, a CAN controller may be “Error Active” (fully operational), “Error Passive” (partly operational), or “Bus Off” (not operational).

An “Error Passive” CAN controller is still able to send messages, even though it has a penalty to wait for a longer bus idle phase, until it is accessing the bus. Also, the “Error Passive” CAN controller is not able to send active Error Frames. This means, that it can see further errors on the bus, but obviously can’t indicate this to other CAN controllers. The “Error Passive” state can be reached, if a certain amount of reception or transmission errors have been detected.

If a CAN controller goes “Bus Off”, then it stops any transmissions or receptions. Depending on its setting, it must either wait on a user (software) trigger to restart again, or it has to perform a recovery sequence. The recovery sequence is a long penalty time, while waiting for several long gaps during the bus communication. Therefore, on a highly loaded bus system, a “Bus Off” node might not be able to reconnect itself to rejoin the communication, if not triggered by the user (software). The “Bus Off” state can be reached only, if a certain higher amount of transmission errors have been detected. Reception errors are not leading to the “Bus Off” state, neither do missing acknowledges (see “**3.3.1 No Acknowledge by another Station (Receiver)**”).

3.3 Error situations of a CAN Transmitter

3.3.1 No Acknowledge by another Station (Receiver)

If a message is sent out correctly without errors, but no other station has acknowledged it, then the CAN Transmitter will flag its message by an Error Frame, and then repeat the message, until it is acknowledged.
However, each unacknowledged message increases the transmit error count; even though a “Bus Off” state cannot be reached by that.
But, when reaching the “Error Passive” state, the error flagging cannot be done any more. Instead, the distance between the messages will increase, because the CAN controller now has a penalty to have to wait a longer time on a free bus condition.

3.3.2 Transmission is not read back

Every CAN Transmitter is permanently monitoring its own transmission by reading back from the bus. If this fails, then this situation is an error condition.

Consequently, a CAN controller needs a CAN bus transceiver, which is providing the read-back functionality. If a transceiver shall not be used (for example, for intra-module communications), then special circuitry is required instead (see "**2.3 CAN Bus without Transceivers**").

Within the arbitration phase of a message, dominant levels may overrule recessive levels. In this case, this is no bit error, but a lost arbitration instead. Then, the CAN controller will stop its current transmission and repeat it, as soon as the CAN bus is free again.
If a sent bit value is not read back outside of the arbitration phase, or if during arbitration phase, a dominant sent bit is received recessive, then this is called a bit error, which will increase the transmit error counter of the CAN controller. Such kind of error can happen by several reasons:

(1) The CAN bus transceiver is not in a valid operation mode
• Activate the CAN bus transceiver or replace it. See "**2. Bus Transceivers and CAN Controller Operation Modes**" for valid combinations of CAN bus transceiver and CAN controller operation modes.

(2) The CAN bus has huge capacitive load or a short-cut.
• Check the CAN bus level voltage in an idle bus state. It should be at 2.5V on both lines.
• Check the CAN bus level voltage during transmissions. The differential voltage between the bus lines increases by at least 1.5V, if dominant bits are on the bus.

(3) Another CAN controller on the same bus has sent an Error Frame (only bit errors of recessive bits).
• Check the bit rate of all CAN controllers on the bus. Easiest way to do this, is to measure the width of an Error Frame on the local TX signal of the CAN controller. It must be 6 bits wide of the expected bit rate.
• Format of the CAN bus is incompatible. Do not combine classical CAN nodes with CAN-FD nodes.

Transmission errors of this kind will cause the CAN controller to go to Error Passive or Bus Off states.

3.4 Error situations of a CAN Receiver

3.4.1 Frame Form Errors

While receiving CAN messages, the frame format is permanently checked if it is in line with the CAN MAC specification. For example, the following format parameters are checked:

• Bit stuffing rules
• Values of dedicated control bits in a message
• Checksum value of a message

During reception, the CAN controller performs hard- and soft-synchronizations. With that, it can adjust to slightly misaligned bit rates. Also, by having a dedicated sampling point position, glitches (spikes) within bits can be filtered.
However, if synchronization cannot be achieved any more, or if disturbances on the bus are getting too heavy, or if there is a node on the CAN bus which is working with an incompatible frame format, then frame form errors will occur.
The receiving CAN controller will react by sending an Error Frame (active or passive, depending on its status).

3.4.2 Reaction on Error Frames of other Nodes

An Error Frame, which is received, is violating the bit stuffing rules. Therefore, a CAN receiver will react on an Error Frame by sending an Error Frame on its own.
In addition, the initiating node of an Error Frame will see the reaction Error Frame of other nodes, and will substitute this Error Frame by another, starting at the first bit of the second Error Frame. Like this, Error Frames get accumulated and partly substituted with a maximum length of two Error Frames plus one bit (13 bits in total).
After this, the bus gets idle again, because every node must keep a recessive delimiter after an Error Frame.
The consequence for a local node is, that due to Error Frames accumulation from other nodes, the internal error counters will increase again.

3.4.3 Acknowledgment Errors

If the acknowledge bit of a CAN receiver gets disturbed, this is a bit error, because the dominant acknowledge bit cannot be turned into a recessive bit, apart from bit error situations. Consequently, this causes an increase of the internal receive error counters and the transmission of an Error Frame.

3.5 Blocking situations for CAN controllers

3.5.1 Permanently dominant blocked CAN bus

A permanently (or a longer time, more than 13 bits) blocked CAN bus on the dominant level is a serious error condition, which disables all communication. Three scenarios exist, which are leading either to “Error Passive”, to the “Bus Off” or “Not Integrated” state of CAN bus nodes.

(1) If a node is just starting, trying to integrate on the CAN bus (setting operational mode):
The dominant level will freeze the integration algorithm, as it is waiting for at least 11 recessive bits. All Renesas CAN controllers apart from RS-CAN, RS-CANFD (all versions), will freeze, until the dominant level has been resolved. Here, a software timeout is required in order to avoid a blocking situation.
RS-CAN and RS-CANFD (all versions) have a detection mechanism integrated, which will raise an interrupt or set a flag, if a blocked dominant bus is detected. This allows software to handle the situation.

(2) If a node is operating, but receiving a message or waiting for a message to receive:
The dominant blocked bus will cause an error detection, which in turn triggers an Error Flag to be sent. After that, the error delimiter will be found to be corrupted. This causes subsequent errors, so that the “Error Passive” state is reached after 24 bits.

(3) If a node is operating and just going to send a message, when the CAN bus is blocked dominant:
The bit errors during transmission will cause that the node reaches “Error Passive” state after 24 bits, and will go “Bus Off” after further 16 bits of dominant blocking situation.

3.5.2 Transceiver in wrong mode or defect

Several CAN transceivers are presenting a permanent dominant signal, if they are triggered to wake up by a CAN bus event. Therefore, proper combinations of operation modes of CAN controllers and transceivers have to be used, in order to avoid blocking situations. See “**2. Bus Transceivers and CAN Controller Operation Modes**” for more information.

A defect CAN transceiver may do the same; but also sporadically or permanently not presenting the CAN bus value in receive direction or falsify transmit levels.

- If a CAN transceiver is not showing all dominant signals on a CAN bus, this will lead to severe communication disturbances on the bus for the whole network, because the CAN controller may assume a free bus, where it is not.
- If a CAN transceiver is not showing all recessive signals on a CAN bus, this will cause local errors only, so that the CAN controller may go “Bus Off” or “Error Passive”.
- If a CAN transceiver is not sending dominant bits, this will cause local errors only, so that the CAN controller may go “Bus Off” or “Error Passive”. This scenario may also happen if a CAN transceiver is in a standby mode, while the CAN controller operates.
- If a CAN transceiver is not sending recessive bits, but dominant instead, this will lead to severe communication disturbances on the bus for the whole network and the local CAN controller, too, which may go “Bus Off” or “Error Passive”.

3.6 Bus Off state: Recovery methods and conditions

When recovering from the “Bus Off” state, at least the following options are possible for Renesas CAN controllers (availability of application for CAN controller types is given in brackets):

(1) Automatic bus off recovery according to ISO 11898-1 [all]
In case of a “Bus Off” error state, the recovery sequence with 128 times of 11 recessive bits is checked, then the CAN controller is in operative mode again.

(2) Bus off recovery according to ISO 11898-1 after software command
[RCAN, RS-CAN, RS-CANFD (all versions) only]
In case of a “Bus Off” error state, the CAN controller stops. After a command by software, the recovery sequence with 128 times of 11 recessive bits is checked, then the CAN controller is in operative mode again.

(3) Immediate recovery to integration state after software command [all except DCAN and FCAN]
In case of a “Bus Off” error state, the CAN controller stops. After a command by software, the CAN controller is restarted in integration state and rejoins the CAN bus after 11 recessive bits.

As DCAN and FCAN controller types only support the method of (1), special a software measure is required for these. Triggered by the Bus Off interrupt, the DCAN and FCAN controllers can be put into initialization mode, where the communication is stopped. After that, option (3) can be implemented by restarting the CAN controller upon software command.

4. Emulation of Efficient Energy Management Concepts of CAN (EEM)

4.1 Abstract

This chapter covers:
- Short introduction in EEM concepts for CAN
- Emulation principles of EEM for CAN without specific hardware (transceiver)
- Requirements for successful emulation of EEM

4.2 Partial Networking

4.2.1 Principle of Operation

In a CAN network, typically all nodes are permanently connected. If the network is in a sleep state, all nodes must be in this state. If at least one node wakes up, its activity on the CAN bus is waking up all other nodes at the network, too.
Regarding effective energy management, it is however desirable, to have certain nodes active, and other nodes in a sleeping state. Therefore, the network waking up behaviour has been improved to become selective.
The legacy wake up pattern (WUP) is a shaped pulse on the network line, but it has no information for a selective wake up method. Therefore, in addtion the wake up frame (WUF) has been declared. The WUF is a CAN frame in classical CAN format (non-FD), whose identifier (ID) and data fields are containing selective wake up information.
Certain assigned nodes of the network would take the role to decide, which parts of the network would wake up, by transmitting the appropriate WUP frames. The WUP frames can also be used to set certain nodes in a sleep state.
Like this, a network configuration may change from “A” to “B”, as shown in the figure below.

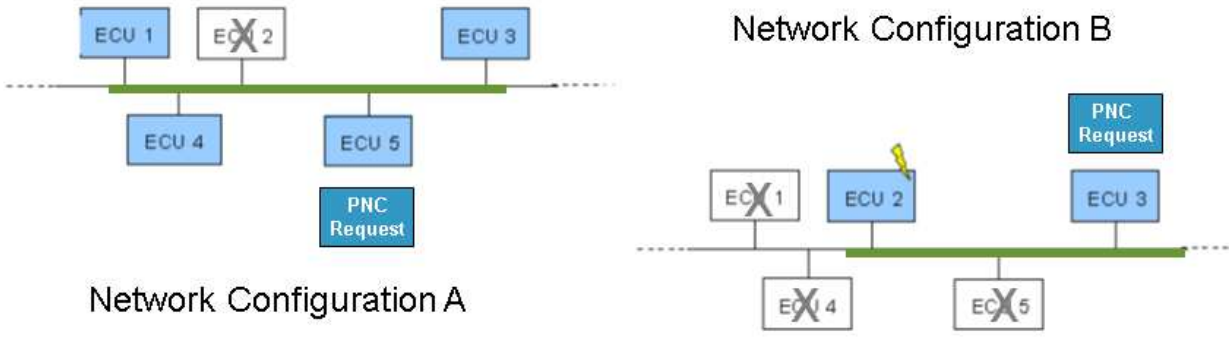

Figure 4.1 Partial Networking Configuration Change

New kinds of CAN transceivers are able to detect both WUF and WUP events on the CAN bus, and they even can decode the WUF content to analyze a local wake up condition. However, these tranceivers are more expensive and need an additional interface to program the wake up conditions.

4.2.2 Requirements for Systems and CAN Controllers

Partial Networking can be emulated with conventional CAN transceivers and CAN controllers, which are having dedicated feature sets. This will avoid the necessity to use a dedicated CAN transceiver for partial networking (ISO 11898-6). Functionality of detection of the WUF will move into the CAN controller, which must stay active even in sleep state of the microcontroller device. Consequently, the following requirements for the microcontroller (MCU) and its integrated CAN controller can be defined:

- A low power mode of the MCU must exist with high power saving effect, where the CAN controller is still active with enough precision (quartz oscillator should run), in order to decode the WUF.
- The used conventional CAN transceiver should provide a low power consumption mode, where it passes the CAN bus signal to the CAN controller (RX), but may have transmission disabled (TX off).

4.2.3 Application on Selective CAN-FD Usage

4.2.3.1 Definition and Requirements

Most legacy CAN controllers are not supporting CAN-FD according to the new ISO 11898-1 standard of 2014 or younger, if the MCU products are older than this new ISO specification.
It is a drawback of CAN-FD, that CAN-FD is not downwards compatible with the classical CAN of the previous ISO 11898-1 standards. This means, legacy classical CAN controllers will disturb the CAN-FD protocol if they are connected to a CAN-FD bus system, so that the communication would brake.

One of the concepts to allow legacy classical CAN controllers on a CAN-FD bus is the usage of Partial Networking. As long as the CAN-FD bus is using the classical CAN communication mode, the legacy nodes may be active; and as soon as the new CAN-FD format is used, the legacy classical CAN nodes would be put into sleep state.

Still, this concept can also be emulated, thus saving a special CAN transceiver of ISO 11898-6, which now also would need to be capable to tolerate the CAN-FD communication.

The drawback of the emulation is, that the energy efficiency will not reach the level of a CAN-FD partial networking transceiver solution. Nevertheless, in some cases this lack of efficiency may be neglectable. For example, if the Partial Networking and CAN-FD usage is performed during system upgrades (software downloads) or similar temporary use cases only.

The concept defines the following requirements:

(1) **The WUF of Partial Networking is a classical CAN frame.**
This is to allow the legacy classical CAN controllers to handle it according to ISO 11898-6, but emulated.

(2) **The WUF and sleep commands of Partial Networking is a highest priority message.**
For fast and timing-precise switching, the Partial Networking messages should win arbitration against all other bus traffic.

(3) **The partial networks must not overlap.**
This means, that the node which performs Partial Networking control by WUF, must do the network switching break in a “break-before-make” manner. So, each switching first must stop the previous communication mode (either nor not CAN-FD), before activating nodes of the other mode. Otherwise, bus errors would occur during the switching phase. As the CAN bus arbitrates, the control of the Partial Networking switching functionality must be done within one single node; for example a central gateway unit.

The emulation of the concept defines the following, additional requirements, in order to work safely:

(4) **When following a CAN-FD frame, a WUF must be sent at least twice.**
If a legacy classical CAN controller emulates Partial Networking, it will listen (-only) to the CAN bus in order to get the WUF. Within CAN-FD phases however, the classical node might get into any error condition, which needs not to be recovered at the end of a CAN-FD frame. As the classical node might need to complete its local error processing (like sending an internal error frame), this processing may overlap a subsequently following frame, invalidating it by that.

(5) **A bus integration phase (CAN bus is recessive for 11 bits) must be inserted after the last WUF.**
When the WUF is received, the emulation of Partial Networking in the local CAN controller requires an operation mode change from “receive-only” into “communication”. Consequently, the CAN controller would restart its transfer layer with the bus integration phase.

(6) **An additional delay must be considered to allow the mode change after a WUF.**
As the local CAN controller changes its mode, apart from the required bus integration, during the mode change, the local CAN controller is not listening to the CAN bus.

4.2.3.2 Principle of Operation in RL78/F1x (or similar with RS-CAN)

The following figures illustrate the operation principle. “RL78/F1x” is a MCU with legacy classical CAN controller, not able to support CAN-FD. It is emulating the Partial Networking, and by this, including the additional requirements as mentioned, it is able to participate on a CAN-FD bus system during classical CAN operation phases.

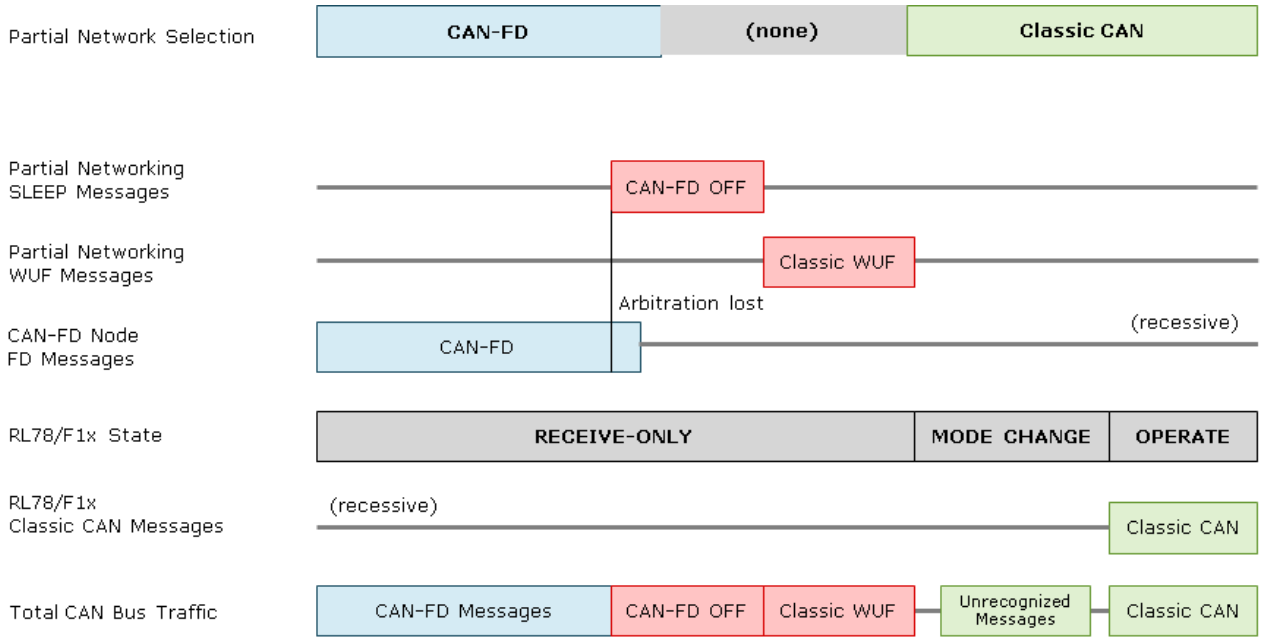

Figure 4.2 Changing Partial Networking to Classical CAN Network operation

Repeating the WUF for the classic CAN is only required, if the “CAN FD OFF” Partial Networking message is not a separate one. In this application, the CAN FD OFF” message is a classical CAN message, and therefore, the WUF frame needs not to be repeated.
The classical CAN controller will need 13 bits of intermission during its integration phase, before it can start communication, plus an additional delay of about 10~50 clocks of its operation clock, in order to perform the mode change. The additional delay length is very much implementation dependent.
It must be avoided that after the mode change, that the CAN bus is at 100% load, because then the required 13 bits of intermission would not be available to allow the integration phase to complete. Maximum bus load should be lower than 90%.

In the application, the same ID is used for both enabling the CAN-FD communication and waking up the the classical CAN node of RL78/F1x. Using ID=0 is representative for the highest priority of the network; this may be adjusted depending on the used network definition.

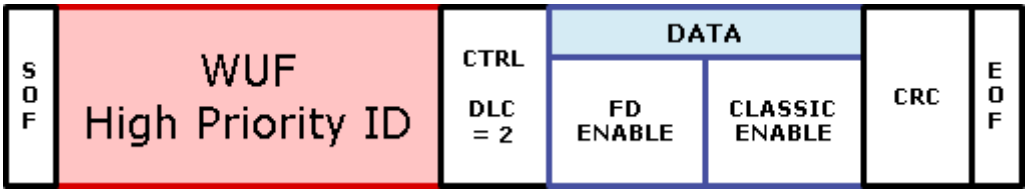

Figure 4.3 WUF Frame for Selective CAN-FD Usage

Using this WUF Frame, the “CAN FD OFF” message is mapped to “FD ENABLE = CLASSIC ENABLE= 0”, and the “classic WUF” message is mapped to “FD ENABLE = 0, CLASSIC ENABLE = 1”. To return to CAN-FD usage, the sequence is vice versa: “classic OFF” is identical with “CAN FD OFF”, and “CAN FD ON” is mapped to “FD ENABLE = 1, CLASSIC ENABLE = 0”.

4.2.3.3 Classical CAN Node Execution Flow in RL78 (or similar with RS-CAN)

The applicaton example provides a simplified flow chart, which shall explain the implementation approach in a generic manner.

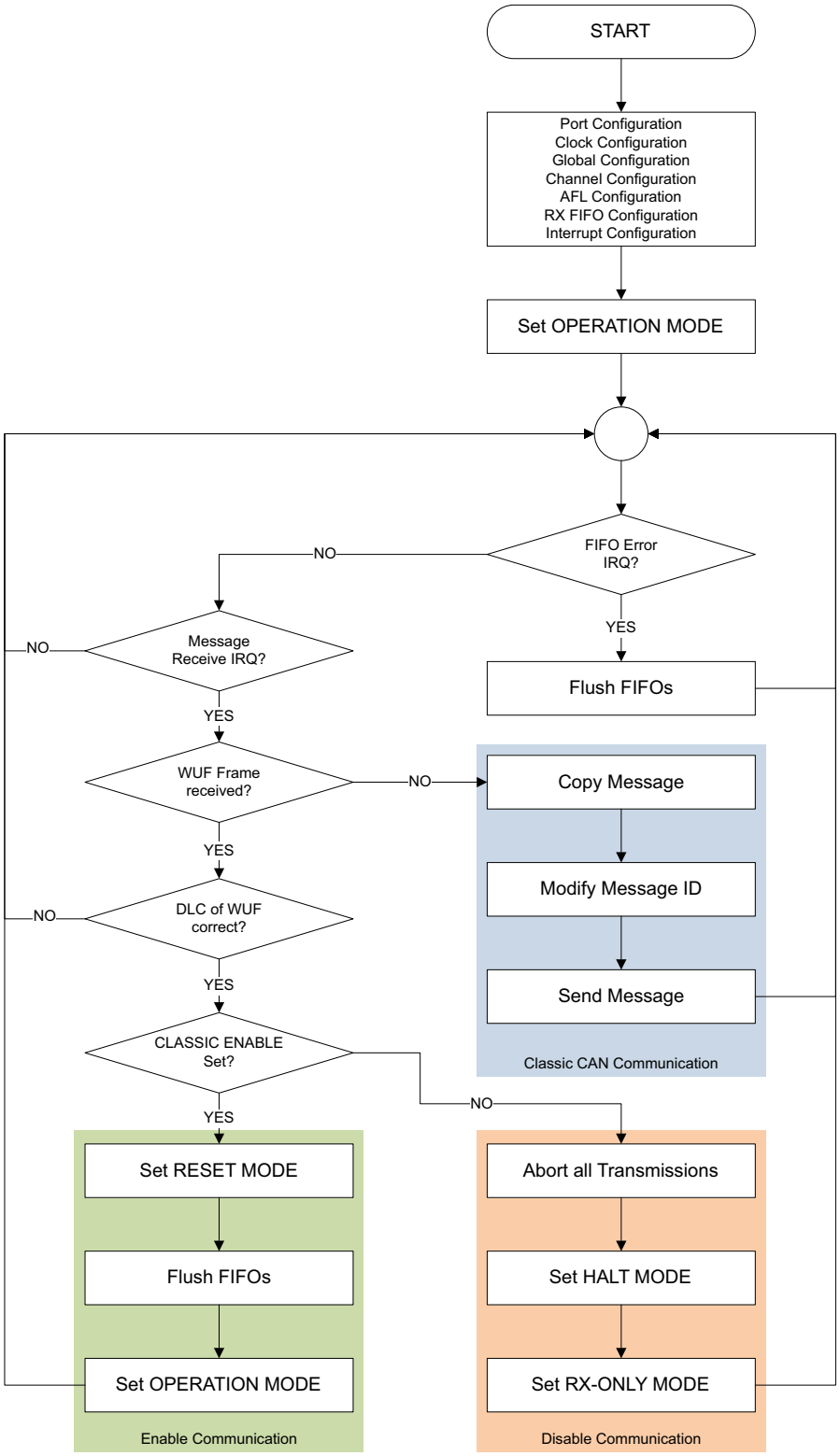

Figure 4.4 Selective CAN-FD by Partial Networking Classical Node Flow Chart

The CAN-FD nodes would require to have a similar flow execution, but by using the “FD ENABLE” flag instead of the “CLASSIC ENABLE” flag.

The classical node’s behaviour in case of its enabled communication is a simple copying and sending back of a received message with a modified identifier. This is highlighted blue in the flow chart, and should be seen as an example behaviour to show the working concept.

To enable the communication if the “CLASSIC ENABLE” flag is set in the WUF message, first the RESET mode of the CAN controller is set. This clears any upcounted errors of the previous operation, which may have been caused by not understood CAN-FD frames.

Note: The CAN controller of RL78/F1x is supporting the RESET mode. For other CAN controllers of Renesas (like AFCAN or FCN), the INIT mode can be used alternatively, in conjunction with a forced clearing of any error states (CCERC).

To disable the communication if the “CLASSIC ENABLE” flag is not set in the WUF message, all pending transmissions of the previous communication are canceled, as far as they are still pending. Then, the HALT mode is set in order to move to the RX-ONLY mode. In the RX-ONLY mode, the CAN controller can receive messages (as far as they are not CAN-FD messages) in order to detect a new WUF, but the CAN controller will not send. This prevents the distortion of the CAN bus by error frames, if CAN-FD messages would appear.

Note: The CAN controller of RL78/F1x is supporting the HALT mode. For other CAN controllers of Renesas (like AFCAN or FCN), the INIT mode can be used alternatively.

4.2.3.4 Renesas CAN Controller Compatibility and Evaluation Result

Renesas CAN controllers of type RS-CANFD (all versions) do not need this application, because they are CAN-FD enabled.

Renesas CAN controllers of type RS-CAN and RS-CANLite are supporting this application and have been evaluated successfully.

Renesas CAN controllers of type AFCAN, DAFCA, FCN and DCN are supporting this application in principle, because they are providing the RX-ONLY mode.

Renesas CAN controllers of type FCAN are supporting this application in principle by using the TPE (Transmit Pin Enable) function to emulate the RX-ONLY mode.

Renesas CAN controllers of type RCAN are supporting this application in principle by using the “Listen-Only-Mode” as RX-ONLY mode.

Renesas CAN controllers of type DCAN are not supporting this application.

4.3 Pretended Networking

4.3.1 Principle of Operation

Pretended Networking is an alternative or secondary way to implement EEM on CAN, thus reducing the power consumption of the overall system. While *Partial* Networking is using dedicated network configuration changes to disable nodes of the network, *Pretended* Networking is based on local decisions and power reduction of individual nodes.
With Pretended Networking, the network configuration typically does not change. Instead, the local CAN nodes may decide to reduce their power consumption and functionality depending on the overall system status, which is derived and interpreted from current network messages. But in any case, a node which uses Pretended Networking, shall not degrade its availability for other nodes; in other words, the local power state shall not affect the CAN communication capabilities of a node.
Even when in a power down mode, the regular operation capability is *pretended* towards other nodes, unless explicitly queried.

4.3.2 Requirements for Systems and CAN Controllers

Pretended Networking typically is not supported by dedicated hardware, because its implementation has too many variations. For this reason, the software emulation of Pretended Networking is the favourite way of implementation in most cases. However, in order to have a real benefit in efficiency, certain requirements for the MCU hardware should be fulfilled.

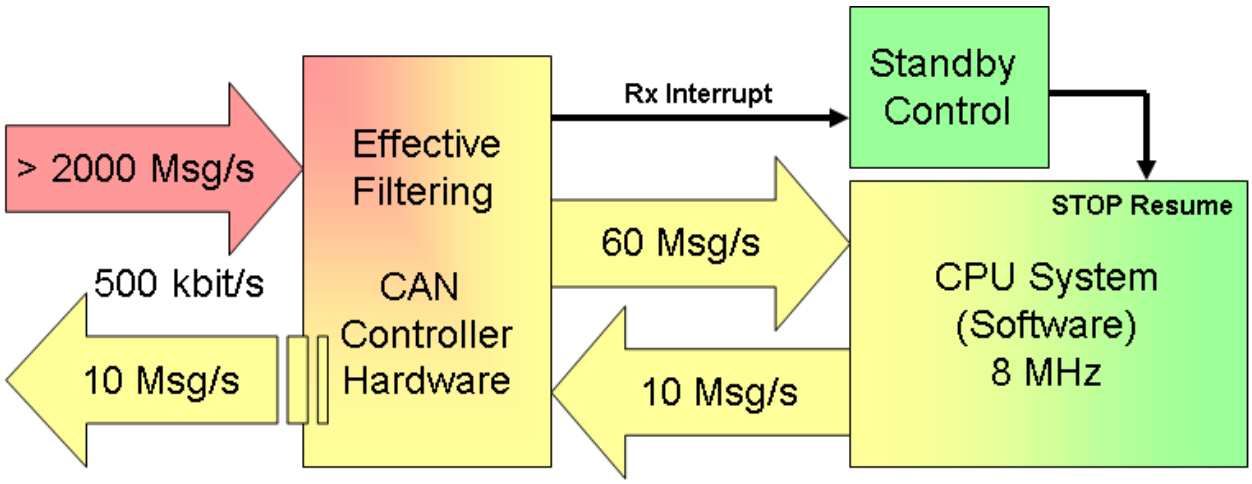

Figure 4.5 Efficient CAN Controller Hardware for Pretended Networking

(1) **The CAN controller should support effective filtering functionality.**
A standard masking principle very often is not sufficient, because when in power down state, the conditions of Pretended Networking to resume full operation may be various. This means, several different messages without a common grouping criteria may be required to check in order to detect a resume condition. If additional filtering must be done by software, the power saving mode must be left for that, so that the efficiency of the power down mode will suffer.
If the CAN controller in addition supports filtering of messages by data content (not just by identifier), this would improve the effectiveness even more.

(2) **The CAN controller should be able to wake the CPU System from a standby mode.**
This also implies that the CAN controller may run (and is still provided with clock), without the CPU system. While Pretended Networking Level 1 simply requires a “HALT” state of the CPU (idle, but clocked), Level 2 of Pretended Networking requires more efficiency in power saving. Even though a suspended CPU will have longer reaction time to resume, the Level 2 of Pretended Networking is designed to support as much power saving as possible.

Note: The Renesas MCU series F1x is providing the appropriate CAN controller and power saving functionality to support Pretended Networking Levels 1 and 2 (by software-emulation of ICOM).

4.4 EEM Concept Comparison

When comparing Partial and Pretended Networking, it turns out that every concept has its advantages and disadvantages. It is possible to combine both concepts in a CAN node, so that depending on the network situation either concept can become effective.

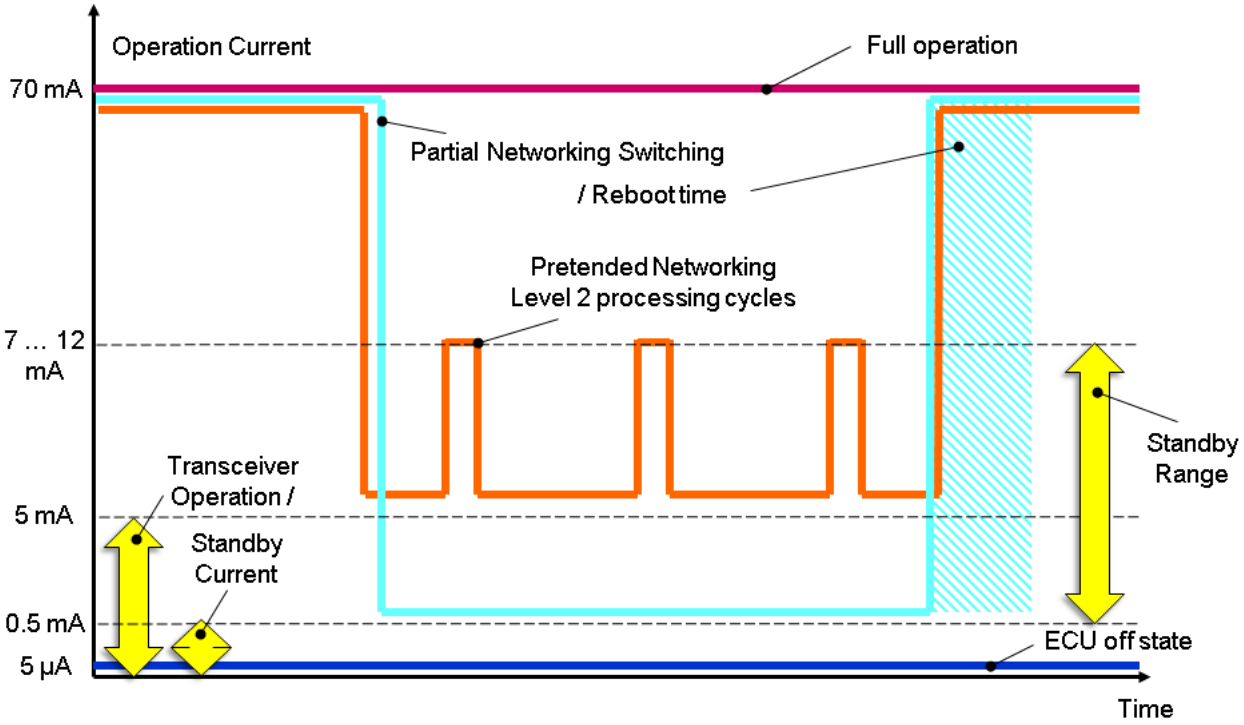

Figure 4.6 Power Consumption and Reaction Time Characteristics of EEM Concepts

Advantage of Partial Networking is lower power consumption, if the EEM concept is active. On the other hand, the resuming time is much longer than for Pretended Networking. When using Pretended Networking, it is vice versa. The current values in the above diagram are examples, showing a “typical” CAN transceiver and MCU in added values.

4.5 Application Examples for Partial and Pretended Networking

Renesas is providing additional information and sample programs for RH850/F1x regarding EEM and CAN controller usage. These can be found with references “R01AN2488EJ”, “R01AN1877ED”.

5. Sample Software Description

5.1 Abstract

This chapter is describing the software packages, which are available upon request as usage examples (so-called “Sample Software”) for the currently available CAN controllers on microcontroller devices.

The software and the description in this chapter are given free of charge, and therefore some conditions apply:
• No guarantee of function for customer specific hardware implementations
• No liability from Renesas side on consequent issues (such as failures, loss of data, injury), caused by its usage
• No allowance to use the software in any commercial product
• No copying of the software without mentioning these conditions
• No announcement of any changes from Renesas side
• No tracking of issues or versioning
• No upgrade compatibility when replaced

The following is covered by this chapter:
• Lower CAN driver functionality
• Mapping of the lower CAN driver to specific device properties
• Applications based on lower CAN driver functions, including a serial or debugger based monitor program

As a general approach, this documentation is not describing each and every step and definition of the software. Instead, we are describing the functionality by showing the algorithmic content, hereby highlighting certain topics which are of certain interest.
All other information the user shall determine from the software programs directly.

The documentation is made for several kinds of CAN controllers, as it is a target to have the API as much close and similar, we are distinguishing among the CAN controllers in sub-chapters. Each CAN controller has a certain prefix, this allows that several software parts for different controller types can be integrated at the same time. The prefix is indicated by <xxx> in this description.
Not all functions are available for all CAN controller types (in case not mentioned, functions are not implemented).

5.2 Supported CAN Controller Hardware

In the current state of this documentation release, the following CAN controller types on device series can be supported:

Table 5.1 Sample software availability on request for CAN Controller Hardware

Device Class	Device Family Members	CAN Controller Type	Prefix <xxx>[a]	Index[b]
78K0R	FJ3	AFCAN ([c])	EE_AFCAN	(A)
RL78	D1A, F12			
V850	FG2, FJ2, SG2, FJ3, FK3, JH3, CARGATE+, -F, -3G, PHOENIX-FS			
	DN4	FCN		(B)
RL78	F13, F14, F15	RS-CANLite	EE_RSCAN	(C)
RH850	F1A, F1L, F1M, F1H, D1M, D1L, P1M	RS-CAN		(D)
	F1K, E2L, E2M, E2H, E2UH	RS-CANFD V2	EE_RSCFD ([d])	(E2|x)
	F1KM	RS-CANFD V3		(E3|x)
	U2A	RS-CANFD V4	RSCFD	(E4)
	P1H-C	M_CAN	MCAN	(F)
		M_TTCAN	MTTCAN	(G)

a. The prefix is used in function names and *#define* constant names of the different drivers of the CAN controller types.
b. Label referenced by function chapters (“implementations”), to show if available/implemented.
c. Sample Software for AFCAN is no longer maintained.
Existing packages can be provided “as is”, but without support and guarantee of operation.
d. Versions V2 and V3 with same API, implementation depending on product. Configuration ranges of V2 (E2) and V3 (E3) are different. Both versions are referenced by (Ex).

5.3 Lower Level CAN Driver Functionality

5.3.1 Overview

Purpose of a lower level CAN driver is to implement the hardware based abstraction into a certain function set, which we will further call “API” (Application Programming Interface).
The main tasks in a CAN driver for this API are:
• Initialization of the CAN controller environment, such as port and clock settings
• Initialization and configuration of the CAN controller
• Starting and stopping of the CAN controller
• Providing means of sending and receiving messages
• Providing means of interrupt processing, including hooks for interrupt handling expansions
• Providing status information of the CAN controller
• Reading and clearing error states of the CAN controller
• Optionally, some self-testing routines

The lower level CAN driver resides in the files *_p.c* with API **_p.h* and definition in **.h*.

5.3.2 Environmental Initialization

Within this driver description, we assume that the clock speed of the CAN controller is fixed and given by some setting of the clock system of a specific device. Therefore, we will have a certain definition constant, which specifies the clock speed, and this constant will have an influence on the bit rate setting, for example.
Other timing we are doing by simple loops with a maximum loop count in order to have a safe way out of hardware failures.
Summarized, for clock settings, the lower CAN driver has no API. The clock speed is a known constant and timeout loops have to be adjusted by specifying the maximum amount of loops (which in fact has a dependency on the used clock speed, of course).
Regarding interrupts, the lower level driver contains a commonly called interrupt routine, which also supports callback mechanisms. By mapping (see 5.4), the interrupt sources are bound with the interrupt controller.
Each CAN channel has two ports to the outside world, which is the CAN *transceiver*. A CAN transceiver is doing the physical layer adaptation from the symmetric CAN bus signal into the *transmit* and *receive* direction ports of the CAN controller. The CAN controller requires that its transmission get visible on the CAN bus, and the CAN bus gets visible (read back) on its reception path. This is provided by the CAN transceiver.
For this reason, the two I/O ports need to become initialized, before starting the CAN controller.

5.3.3 Used Types

The following data types are used in the API, which are bound to elementary ANSI C data types. Parameters and functions are named with corresponding endings in order to indicate the data type.

Table 5.2 Used Types in API

Data Type	Ending	ANSI Type	Description
u08	_u08	unsigned char	8 bits
s08	_s08	signed char	7 bits plus sign bit at MSB position
u16	_u16	unsigned short	16 bits
s16	_s16	signed short	15 bits plus sign bit at MSB position
u32	_u32	unsigned long	32 bits
s32	_s32	signed long	31 bits plus sign bit at MSB position
flt	_flt	float	floating point decimal
dbl	_dbl	double	floating point double precision decimal
pu08	_pu08	unsigned char *	pointer to u08 (pass by reference)
pu16	_pu16	unsigned short *	pointer to u16 (pass by reference)
pu32	_pu32	unsigned long *	pointer to u32 (pass by reference)
bit	_bit	enum {false, true}	boolean enumeration with false = 0, true = 1
[pointer to structure]	-	-	see documentation of the function
void	-	void	function returns no value

5.3.4 Port I/O Initialization

5.3.4.1 <xxx>_PortEnable( )

Implementations: (A), (B), (C), (D), (Ex), (E4), (F), (G).

(1) All Implementations

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
ChannelNumber_u08:	Selected CAN Controller channel [(C), (D), (Ex), (E4)]
MachineNumber_u08:	Selected CAN Controller channel [(A), (B)]

(1-2) Return Values

<*xxx*>*_ERROR* on parameter failures, otherwise <*xxx*>*_OK*.

(1-3) Functional Description

The function enables (activates) the port I/O for the dedicated CAN controller unit and channel (if several channels are available for a unit).
A standard port support library is called to perform the function.
For each used CAN controller unit and channel, the following dedicated *#define* constants must be set in the driver's mapping definition (see 5.4 for details and file information):

#define <xxx>_PORT_M<*unit*>RX<*channel*>	PORT*_p*
#define <xxx>_PORT_BIT_M<*unit*>RX<*channel*>	BIT*_r*
#define <xxx>_PORT_FUNC_M<*unit*>RX<*channel*>	PORT_FUNCTION_ALTLV*_a*
#define <xxx>_PORT_M<*unit*>TX<*channel*>	PORT*_q*
#define <xxx>_PORT_BIT_M<*unit*>TX<*channel*>	BIT*_t*
#define <xxx>_PORT_FUNC_M<*unit*>TX<*channel*>	PORT_FUNCTION_ALTLV*_b*

These settings will bind the port I/O lines *p.r* and *q.t* in the alternate port assignment level *a* vs. *b* to the CAN controller unit <*unit*>, channel <*channel*>; where *p, q, r, t, a* and *b* are decimal text characters (for numeric ports and port bits of the specific device).

5.3.4.2 <xxx>_PortDisable( )

Implementations: (A), (B), (C), (D), (Ex), (E4), (F), (G).

(1) All Implementations

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
ChannelNumber_u08:	Selected CAN Controller channel [(C), (D), (Ex), (E4)]
MachineNumber_u08:	Selected CAN Controller channel [(A), (B)]

(1-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(1-3) Functional Description

The function disables (deactivates) the port I/O for the dedicated CAN controller unit and channel (if several channels are available for a unit).
A standard port support library is called to perform the function.
The *#define* constants used for this function are the same as for function <xxx>_PortEnable( ).

5.3.5 CAN Controller Initialization and Configuration

Even though there is the intention to have the same way of initialization for all kinds of CAN controllers, this is not feasible always. The way of initialization for each CAN controller type by calling the following functions in the right way is described in chapter 5.5.

5.3.5.1 <xxx>_SetGlobalConfiguration( )

Implementations: (A), (B), (C), (D), (Ex), (E4).

(1) Implementations: (A), (B)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
MainClockPrescaler_u08:	Main clock prescaler setting
ABTDelay_u08:	Message delay setting for ABT mode [(A) only]

(1-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(1-3) Functional Description

Explicit parameters are used to set the global configuration.
The global configuration is setting the global clock prescaler, test functionality and other global parts of the functionality.
The function also waits on any hardware initialization phases after *reset* and releases the global soft reset of the CAN controller.

(2) Implementations: (C), (D), (Ex), (E4)

(2-1) Parameters

UnitNumber_u08:	Selected CAN Controller
Config:	Global configuration structure

(2-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(2-3) Functional Description

Parameters are passed by using referencing a global configuration structure.
The global configuration is setting the global clock used for timestamps, global CAN-FD options, test functionality, memory configuration (partitioning) and other global parts of the functionality.
The function also waits on any hardware initialization phases after *reset* and releases the global soft reset of the CAN controller.

The global configuration structure <*xxx*>_*cfg_global* contains the following elements (when referring to the structure element's names, further information is available in the corresponding user's manual of the product):

Table 5.3 Global Configuration Structure Elements (Ex)

Element	Type	Value Range	Description
gcfg [GCFG Register]	<xxx>_c_gcfg.tpri <xxx>_c_gcfg.dce <xxx>_c_gcfg.dre <xxx>_c_gcfg.mme <xxx>_c_gcfg.dcs <xxx>_c_gcfg.cmpoc <xxx>_c_gcfg.eefe <xxx>_c_gcfg.tmtsce <xxx>_c_gcfg.tsp <xxx>_c_gcfg.tsss <xxx>_c_gcfg.tsbtcs <xxx>_c_gcfg.itrcp	{0, 1} {0, 1} {0, 1} {0, 1} {0, 1} {0, 1} {0, 1} {0, 1} n ∈ {0 ,,, 15} {0, 1} {0 ... 7} {0, 65535}	Transmission priority DLC check enable DLC replacement enable Mirror mode enable Communication clock selection ([a]) Message payload overflow configuration ([b]) ECC error flag enable ([c]) Transmit timestamp capture enable ([d]) Timestamp prescaler; value is 2^n Timestamp global clock select (bit time or global) Timestamp channel bit time clock select FIFO interval timer prescaler
gctr [GCTR Register]	<xxx>_c_gctr.gmdc <xxx>_c_gctr.gslpr <xxx>_c_gctr.ie <xxx>_c_gctr.tsrst <xxx>_c_gctr.tswr	- - ([e]) {0, 1} {0, 1}	Always set to *<xxx>_OPMODE_KEEP.* Not used during global configuration. Global interrupt activation Reset timestamp counter Allow explicit setting of the timestamp counter ([f])
gfdcfg [GFDCFG Register ([g])]	<xxx>_c_gfdcfg.rped <xxx>_c_gfdcfg.tsccfg	{0, 1} {0, 1, 2}	Protocol exception event detection enable Timestamp capturing point *<xxx>_TSCAPTURE...* *SOF:* At SOF; *FRVALID:* At EOF; *RES:* At Res Bit
gcrccfg [GCRCCFG Register ([h])]	<xxx>_c_gcrccfg.nie	{0, 1}	ISO protocol compliance *<xxx>_PROTOCOL_ISO* or *<xxx>_PROTOCOL_BOSCHV1 (non-ISO)*
rmnb [RMNB Register]	u32 <xxx>_c_rmnb.nrxmb <xxx>_c_rmnb.rmpls	{0, ... } {0, ... } {0, ... }	Number of used standard message boxes ([i]) Number of used standard message boxes ([j]) Size of standard message boxes ([k])
rnc [RNC Register set as an array]	u32[ ]	{0, ... } for each channel	Amount of reception rules per channel ([l])
rfcc [RFCC Register set as an array]		-	Not used within <xxx>_SetGlobalConfiguration( ). See <xxx>_SetGlobalFIFOConfiguration( ) for initialization.
cdtct [CDTCT Register ([m])]	<xxx>_c_cdtct.rfdmae <xxx>_c_cdtct.cfdmae	{0, 1} for each FIFO; bit position corresponds with FIFO / channel number	DMA Transfer Enable for Receive FIFO [7:0] DMA Receive Transfer Enable for Multi-purpose FIFO#0 of Channel [7:0]
cdttct [CDTTCT Register ([n])]	<xxx>_c_cdtt.tq0dma <xxx>_c_cdtt.tq3dma <xxx>_c_cdtt.cfdma	{0, 1} for each Queue or FIFO; bit position corresponds with channel number	DMA Transfer Enable for TX-Queue#0 of Channel DMA Transfer Enable for TX-Queue#3 of Channel DMA Transfer Enable for TX-FIFO#2 of Channel ... [7:0]
gfcmc [GFCMC Register ([o])]	<xxx>_c_gfcmc.flxc	{0, 1} bit position corresponds with channel pair [n, n+1]	Activate Flexible CAN Mode with channel pair. <xxx>_FLEXCAN_CH0_1, <xxx>_FLEXCAN_CH2_3, <xxx>_FLEXCAN_CH4_5, <xxx>_FLEXCAN_CH6_7
gftbac [GFTBAC Register ([p])]	<xxx>_c_gftbac.flxmb01 <xxx>_c_gftbac.flxmb23 <xxx>_c_gftbac.flxmb45 <xxx>_c_gftbac.flxmb67	<xxx>_FLEXBUF_*n* n ∈ {4|8|12|16|20|24|28|32}	Number of TX Buffers which the even channel is borrowing from the odd channel.

a. When set to *<xxx>_CLOCK_SYS*, the communication clock is automatically adjusted to half of the peripheral clock of the CAN controller. When set to *<xxx>_CLOCK_MOSC*, the direct main oscillator (low jitter) clock is used instead for communication. Here, the user has to take care that the direct main oscillator clock is not faster than half of the peripheral clock.

b. Only available in (Ex) controllers.

c. Only available in (E2) controllers.
d. Only available in (E2) controllers.
e. Use either *<xxx>_GINT_NONE, <xxx>_GINT_DLCCHECK, <xxx>_GINT_MSGLOST, <xxx>_GINT_THLLOST, <xxx>_GINT_FDMSGOVF* or a combination by *binary or* of these for (Ex).
In addition, <xxx>_GINT_GWTXQOVR, <xxx>_GINT_GWTXQLOST, <xxx>_GINT_GWFIFOOVR for (E4).
f. Only available in (Ex) controllers.
g. Only available in (Ex), (E4) controllers.
h. Only available in (Ex) controllers.
i. For (C) and (D) controllers.
See user's manual for the maximum allowed value (depends on product).
j. For (Ex), (E4) controllers.
See user's manual for the maximum allowed value (depends on product).
Consider maximum amount of available storage when setting this value.
k. Only available in (Ex), (E4) controllers. 0: 8 Bytes, 1: 12 Bytes, 2: 16 Bytes, 3: 20 Bytes - (E2), (E3), (E4) additionally: 4: 24 Bytes, 5: 32 Bytes, 6: 48 Bytes, 7: 64 Bytes.
l. Total amount of rules for all channels must not exceed the limit as given in the user documentation.
(C): Max. 16 per channel
(D) : Max. 128 per channel
(E2): Max. 127 per channel
(E3): Max. 255 per channel
(E4): Max. 384 per channel
m. Only available in (E4) controllers.
n. Only available in (E4) controllers.
o. Only available in (E4) controllers.
p. Only available in (E4) controllers.

5.3.5.2 <xxx>_SetGlobalFIFOConfiguration( )

Implementations: (C), (D), (Ex), (E4).

(1) Implementations: (C), (D), (Ex), (E4)

(1-1) Parameters

UnitNumber_u08: Selected CAN Controller
Config: Global configuration structure

(1-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(1-3) Functional Description

Parameters are passed by using referencing the same global configuration structure as used in <xxx>_SetGlobalConfiguration( ). By <xxx>_SetGlobalFIFOConfiguration( ), the part of global Receive FIFO setup is performed in this separate step. The reason for this approach is, that global Receive FIFO setup needs to be done after the RS-CAN(-FD) controller is already initialized in a global operation state.

From the global configuration structure <*xxx*>_*cfg_global*, the remaining elements are used, as shown in the table below:

Table 5.4 Global Configuration Structure Receive FIFO Initialization Elements

Element	Type	Value Range	Description
rfcc [RFCC Register set as an array]	<xxx>_c_rfcc.rfe	-	Not used during FIFO configuration
	<xxx>_c_rfcc.rfie	{0, 1}	FIFO Receive Interrupt enable
	<xxx>_c_rfcc.rfpls	{0 ... 7}	Size of FIFO receive objects (a)
	<xxx>_c_rfcc.rfdc	{0 ... 7}	Depth of FIFO (b)
	<xxx>_c_rfcc.rfim	{0, 1}	Interrupt mode: <xxx>_FIFO_INT_ONLEVEL: use level of rfigcv <xxx>_FIFO_INT_ONEVERY: on every message
	<xxx>_c_rfcc.rfigcv	n ∈ {0 ... 7}	Fill level of interrupt generation (see rfim) (c)
	<xxx>_c_rfcc.rffie	{0, 1}	Receive FIFO Full Interrupt enable (d)

a. Only available in (Ex) controllers.
0: 8 Bytes, 1: 12 Bytes, 2: 16 Bytes, 3: 20 Bytes, 4: 24 Bytes, 5: 32 Bytes, 6: 48 Bytes, 7: 64 Bytes.
Consider maximum amount of available storage when setting this value (see user's manual).

b. 0: 0 Messages (FIFO is disabled), 1: 4 Messages, 2: 8 Messages, 3: 16 Messages, 4: 32 Messages, 5: 48 Messages, 6: 64 Messages, 7: 128 Messages
Consider maximum amount of available storage when setting this value (see user's manual).

c. Value *n* sets the fill level of generating an interrupt to (*n*+1)/8.
Use constants <xxx>_FIFO_ILEVEL_<*n+1*>D8 to set the level.

d. Only available in (E4) controllers.

5.3.5.3 <xxx>_Set[...]Configuration( )

Implementations: (A), (B), (C), (D), (Ex), (E4), (F), (G).

(1) Implementations: (A), (B)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
MachineNumber_u08:	Selected CAN Channel of Controller
BitratePrescaler_u08:	Division factor of bit rate prescaler (use zero for 1:1, 1 for 1:2 etc.)
Segment1Time_u16:	Length of bit time segment 1 in TQ minus one
Segment2Time_u16:	Length of bit time segment 2 in TQ minus one
SyncJumpWidth_u16:	Synchronization jump width in TQ

(1-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(1-3) Functional Description

For these controller types, the function name is <xxx>_SetMachineConfiguration().
Explicit parameters are used to set the channel configuration.
The channel configuration is setting the bit rate prescaler and bit timing settings of the protocol engine:
Time segment sizes in time quanta, synchronization jump width in time quanta.
These parameters must be provided by the user explicitly for correct bit rate setting. The function does not include any automatic bit timing setting functionality.

(2) Implementations: (C), (D)

(2-1) Parameters

UnitNumber_u08:	Selected CAN Controller
ChannelNumber_u08:	Selected CAN Channel of Controller
Config:	Channel configuration structure

(2-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(2-3) Functional Description

The function is named <xxx>_SetChannelConfiguration( ).

Parameters are passed by using referencing a channel configuration structure.
The channel configuration is setting the bit rate parameters either automatically (if bit rate is specified) or by explicit parameters for the bit timing in time quanta. Also, the sampling point setting has a default value of *<xxx>_BT_SPTOPTIMUM*, if no explicit time segment settings are specified.
In addition, the function defines several channel specific settings, default interrupt enable for transmit boxes, transmit queue settings (RS-CAN only), transmit history list settings and Multi-Purpose FIFO settings (see table below).

The channel configuration structure *<xxx>_cfg_channel* contains the following elements (when referring to the structure element's names, further information is available in the corresponding

user's manual of the product):

Table 5.5 Channel Configuration Structure Elements

Element	Type	Value Range	Description
bitrate [Bit Rate]	u32	bits per second 0 ... 1000000	Bit rate to be set for communication. Set to 0, if automatic bit timing setting is *not* desired.
tq_perbit [Bit Resolution]	u08	Time Quanta 5 ... 25	Amount of time quanta per bit to be set for communication. Set to 0, if no preferred setting.
syncjumpwidth [Synchronization]	u08	Sync Time Quanta 1 .. 4	Synchronization Jump Width to be set for communication. Set to 0, if maximum is required.
samplingpointpos [Sampling Point]	flt	Position in Bit (%/100) 0.5 ... 0.96	Sampling Point to be set for communication. Set to 0.0, if preferred default setting is acceptable.
ctr [Channel CTR Register]	<xxx>_c_ctr.chmdc	{0, 1, 2}	Channel state during configuration ([a]). Use either <xxx>_OPMODE_OPER (operation mode), <xxx>_OPMODE_RESET (reset mode) or <xxx>_OPMODE_HALT (halt mode).
	<xxx>_c_ctr.cslpr	-	Not used during channel configuration.
	<xxx>_c_ctr.rtbo	{0, 1}	Force to return from bus-off, if set to 1.
	<xxx>_c_ctr.ie	{one or several flags}	Interrupt sources of channel to be enabled. ([b])
	<xxx>_c_ctr.bom	{0, 1, 2, 3}	Bus off recovery specification (4 options) ([c])
	<xxx>_c_ctr.errd	{0, 1}	First error blocks further reports, if not set.
	<xxx>_c_ctr.ctme	{0, 1}	Test mode enable flag (if set, see *ctms*)
	<xxx>_c_ctr.ctms	{1, 2, 3}	Test mode selection ([d])
	<xxx>_c_ctr,trwe	{0, 1}	Allows writing of error counters, if set
	<xxx>_c_ctr.trh	{0, 1}	Stops error counters, if set
	<xxx>_c_ctr.trr	{0, 1}	Clears error counters, if set
tmiec [TX Interrupt Enable flags]	u16	{0, 1} for each TX buffer; bit position corresponds with buffer number	Enable transmit interrupt or transmit abort interrupt for transmit buffer(s), if set.
txqcc ([e]) [TX Queue configuration]	<xxx>_c_txqcc.qe	{0, 1}	Enable TX Queue, if set.
	<xxx>_c_txqcc.dc	{0, 2 ... 15}	Depth of TX Queue (off, 3 to 16 messages).
	<xxx>_c_txqcc.ie	{0, 1}	TX Queue interrupt enabled, if set.
	<xxx>_c_txqcc.im	{0, 1}	TX Queue interrupt mode. ([f])
thlcc [Transmit History configuration]	<xxx>_c_thlcc.thle	{0, 1}	Enable Transmit History, if set.
	<xxx>_c_thlcc.ie	{0, 1}	Transmit History interrupt enabled, if set.
	<xxx>_c_thlcc.im	{0, 1}	Transmit History interrupt mode ([g]).
	<xxx>_c_thlcc.dte	{0, 1}	Specification what gets an entry in the list ([h]).
cfcc [Multi-Purpose FIFO configuration]		-	Not used within <xxx>_Set[...]Configuration( ). See <xxx>_SetCOMFIFOConfiguration( ) for initialization.

a. The setting is not persistent and used only during the configuration phase. If no special requirement is given, it is recommended to set this to <xxx>_OPMODE_RESET. After configuration, the state will be <xxx>_OPMODE_HALT.

b. Channel interrupt sources are:
<xxx>_CINT_BUSERR, <xxx>_CINT_WARNING, <xxx>_CINT_PASSIVE, <xxx>_CINT_BUSOFF, <xxx>_CINT_RECOVERY, <xxx>_CINT_OVERLOAD, <xxx>_CINT_BUSLOCK, <xxx>_CINT_ARBLOST.

c. Channel Bus Off Recovery options are:
<xxx>_BOM_ISO (according to ISO), <xxx>_BOM_HALTBOFF (HALT mode before recovery), <xxx>_BOM_HALTRECV (HALT mode after recovery), <xxx>_BOM_SW (recovery control by software, no ISO recovery phase).

d. Set *chmdc* to <xxx>_OPMODE_HALT, when using test modes. Test modes are:
<xxx>_TEST_RXONLY (Receive-Only operation mode), <xxx>_TEST_EXTLOOP (external loop self-test mode - includes transceiver), <xxx>_TEST_INTLOOP (internal loop self-test mode - excludes transceiver).

e. This configuration setting is not available for RS-CANLite controller types.

f. Use either <xxx>_TXQ_INT_ONEVERY (every transmission causes an interrupt) or <xxx>_TXQ_INT_ONLAST (interrupt triggered when queue is empty)

g. Use either <xxx>_THL_INT_ONEVERY (every new entry causes an interrupt) or <xxx>_THL_INT_ONLEVEL (interrupt triggered when history list has a fill level of 3/4)

h. Use either <xxx>_THL_ENTRY_ALL (all transmission sources will cause an entry) or <xxx>_THL_ENTRY_QUEUED (only FIFOs and TX Queues will cause entries)

(3) Implementations: (Ex)

(3-1) Parameters

UnitNumber_u08: Selected CAN Controller
ChannelNumber_u08: Selected CAN Channel of Controller
Config: Channel configuration structure

(3-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(3-3) Functional Description

The function is named <xxx>_SetChannelConfiguration( ).

Parameters are passed by using referencing a common configuration structure.
The common configuration is setting the bit rate parameters either automatically (if bit rate is specified) or by explicit parameters for the bit timing in time quanta. Also, the sampling point setting has a default value of <*xxx*>_*BT_SPTOPTIMUM*, if no explicit time segment settings are specified.

The channel configuration structure <*xxx*>_*cfg_channel* contains the following elements (when referring to the structure element's names, further information is available in the corresponding user's manual of the product):

Table 5.6 Channel Configuration Structure Elements

Element	Type	Value Range	Description
arb_bitrate [Arbitration Bit Rate]	u32	bits per second 0 ... 1000000	Bit rate to be set for communication arbitration phase in CAN-FD or in general for classic CAN. Set to 0, if automatic bit timing setting is *not* desired.
arb_ samplingpointpos [Sampling Point]	flt	Position in Bit (%/100) 0.5 ... 0.98	Sampling Point to be set for communication arbitration phase in CAN-FD or in general for classic CAN. Set to 0.0, if preferred default setting is acceptable.
data_bitrate [Arbitration Bit Rate]	u32	bits per second 0 ... 20000000 ([a])	Bit rate to be set for communication data phase in CAN-FD. Set to 0, if automatic bit timing setting is *not* desired.
data_ samplingpointpos [Sampling Point]	flt	Position in Bit (%/100) 0.5 ... 0.93	Sampling Point to be set for communication to be set for communication data phase in CAN-FD. Set to 0.0, if preferred default setting is acceptable.
ncfg [Arbitration Bit Timing] ([b])	<xxx>_c_ncfg.nbrp <xxx>_c_ncfg.nsjw <xxx>_c_ncfg.ntseg1 <xxx>_c_ncfg.ntseg2	{0 ... 1023} {0 ... 31} {1 ... 127} {1 ... 31}	Bit Rate Prescaler for Arbitration. Synchronization Jump Width for Arbitration. Time Segment 1 for Arbitration. Time Segment 2 for Arbitration.
dcfg [Data Bit Timing] ([c])	<xxx>_c_dcfg.dbrp <xxx>_c_dcfg.dtseg1 <xxx>_c_dcfg.dtseg2 <xxx>_c_dcfg.dsjw	{0 ... 127} {1 ... 15} {1 ... 7} {0 ... 7}	Bit Rate Prescaler for Data Phase. Time Segment 1 for Data Phase. Time Segment 2 for Data Phase. Synchronization Jump Width for Data Phase.

Table 5.6 Channel Configuration Structure Elements

Element	Type	Value Range	Description
ctr [Channel CTR Register]	<xxx>_c_ctr.chmdc	{0, 1, 2}	Channel state during configuration ([d]). Use either <xxx>_OPMODE_OPER (operation mode), <xxx>_OPMODE_RESET (reset mode) or <xxx>_OPMODE_HALT (halt mode).
	<xxx>_c_ctr.cslpr	-	Not used during channel configuration.
	<xxx>_c_ctr.rtbo	{0, 1}	Force to return from bus-off, if set to 1.
	<xxx>_c_ctr.ie	{one or several flags}	Interrupt sources of channel to be enabled. ([e])
	<xxx>_c_ctr.bom	{0, 1, 2, 3}	Bus off recovery specification (4 options) ([f])
	<xxx>_c_ctr.errd	{0, 1}	First error blocks further reports, if not set.
	<xxx>_c_ctr.ctme	{0, 1}	Test mode enable flag (if set, see *ctms*)
	<xxx>_c_ctr.ctms	{1, 2, 3}	Test mode selection ([g])
	<xxx>_c_ctr,trwe	{0, 1}	Allows writing of error counters, if set ([h])
	<xxx>_c_ctr.trh	{0, 1}	Stops error counters, if set ([i])
	<xxx>_c_ctr.trr	{0, 1}	Clears error counters, if set ([j])
	<xxx>_c_ctr.crct	{0, 1}	CRC error test activation, if set ([k])
	<xxx>_c_ctr.rom	{0, 1}	Restricted operation mode, if set
fdctr [CAN-FD Control Settings]	<xxx>_c_fdctr.eocclr	{0, 1}	Clear error occurrence counter, if set
	<xxx>_c_fdctr.socclr	{0, 1}	Clear successful occurrence counter, if set
fdcfg [CAN-FD Configuration Settings]	<xxx>_c_fdcfg.eoccfg	{0, 1, 2, 4, 5, 6}	Error occurrence counter configuration ([l])
	<xxx>_c_fdcfg.tdcoc	{0, 1}	Transceiver delay compensation offset config. ([m])
	<xxx>_c_fdcfg.tdce	{0, 1}	Transceiver delay compensation enabled, if set ([n])
	<xxx>_c_fdcfg.esic	{0, 1}	Error state indication configuration ([o])
	<xxx>_c_fdcfg.tdco	{0 ... 127}	Transceiver delay compensation offset in communication clock cycles, rounded to full TQ.
	<xxx>_c_fdcfg.gwen	{0, 1}	Multi-gateway function enabled, if set ([p])
	<xxx>_c_fdcfg.gwfdf	{0, 1}	Multi-gateway function configuration for FDF ([q])
	<xxx>_c_fdcfg.gwbrs	{0, 1}	Multi-gateway function configuration for BRS ([r])
	<xxx>_c_fdcfg.tmme	{0, 1}	TX Buffer merged, if set ([s])
	<xxx>_c_fdcfg.fdoe	{0, 1}	CAN-FD only mode enabled, if set ([t])
	<xxx>_c_fdcfg.refe	{0, 1}	Enable RX edge filter, if set ([u])
	<xxx>_c_fdcfg.cloe	{0, 1}	Classical CAN only mode, if set ([v])
tmiec [TX Interrupt Enable flags]	u16	{0, 1} for each TX buffer; bit position corresponds with buffer number	Enable transmit interrupt or transmit abort interrupt for transmit buffer(s), if set.
txqcc ([w]) [TX Queue configuration, as an array for (E3)]	<xxx>_c_txqcc.qe	{0, 1}	Enable TX Queue, if set.
	<xxx>_c_txqcc.dc	{0, 2 ... 15 | 31}	Depth of TX Queue (off, 3 to 16 (E2) or 32 (E3) messages).
	<xxx>_c_txqcc.ie	{0, 1}	TX Queue interrupt enabled, if set.
	<xxx>_c_txqcc.im	{0, 1}	TX Queue interrupt mode. ([x])
thlcc [Transmit History configuration]	<xxx>_c_thlcc.thle	{0, 1}	Enable Transmit History, if set.
	<xxx>_c_thlcc.ie	{0, 1}	Transmit History interrupt enabled, if set.
	<xxx>_c_thlcc.im	{0, 1}	Transmit History interrupt mode ([y]).
	<xxx>_c_thlcc.dte	{0, 1}	Specification what gets an entry in the list ([z]).
cfcc [Multi-Purpose FIFO configuration]		-	Not used within <xxx>_Set[...]Configuration( ). See <xxx>_SetCOMFIFOConfiguration( ) for initialization.

a. The maximum value depends on the capabilities of the hardware implementation (clock speed).

b. Values in this register are setting the configuration with one more than programmed. These values are used, if automatic bit timing setting is suppressed by specifying 0 or 0.0 for arbitration bitrate / sampling point.

c. Values in this register are setting the configuration with one more than programmed. These values are used, if automatic bit timing setting is suppressed by specifying 0 or 0.0 for data bitrate / sampling point.

d. The setting is not persistent and used only during the configuration phase. If no special requirement is given, it is recommended to set this to <xxx>_OPMODE_RESET. After configuration, the state will be <xxx>_OPMODE_HALT.

e. Channel interrupt sources are:
<xxx>_CINT_BUSERR, <xxx>_CINT_WARNING, <xxx>_CINT_PASSIVE, <xxx>_CINT_BUSOFF, <xxx>_CINT_RECOVERY, <xxx>_CINT_OVERLOAD, <xxx>_CINT_BUSLOCK, <xxx>_CINT_ARBLOST.

f. Channel Bus Off Recovery options are:
<xxx>_BOM_ISO (according to ISO), <xxx>_BOM_HALTBOFF (HALT mode before recovery), <xxx>_BOM_HALTRECV (HALT mode after recovery), <xxx>_BOM_SW (recovery control by software, no ISO recovery phase).

g. Set *chmdc* to <xxx>_OPMODE_HALT, when using test modes. Test modes are:
<xxx>_TEST_RXONLY (Receive-Only operation mode), <xxx>_TEST_EXTLOOP (external loop self-test mode - includes transceiver), <xxx>_TEST_INTLOOP (internal loop self-test mode - excludes transceiver).

h. For (E2) controllers only.

i. For (E2) controllers only.

j. For (E2) controllers only.

k. For (E3) controllers only.

l. User either <xxx>_EOC_ALLTXRX (all frames), <xxx>_EOC_ALLTX (all transmitter frames), <xxx>_EOC_ALLRX (all receiver frames), <xxx>_EOC_ALLTXRXFD (only CAN-FD frames), <xxx>_EOC_ALLTXFD (only CAN-FD transmitter frames) or <xxx>_EOC_ALLRXFD (only CAN-FD receiver frames)

m. Use either <xxx>_TDC_OFFSETONLY (to disable measured offset) or <xxx>_TDC_MEASOFFSET (to combine measured offset with additional offset added by soft setting).

n. Use either <xxx>_TDC_ENABLE (to enable) or <xxx>_TDC_DISABLE (to disable).

o. Use either <xxx>_ESI_BYNODE (to set ESI by local node status) or <xxx>_ESI_BYACTBUFFER (to set ESI by software within TX buffers, if the node is error active).

p. Use either <xxx>_MULTIGW_ENABLE (activate HW multi-gateway function) or <xxx>_MULTIGW_DISABLE (disable HW multi-gateway function). The multi-gateway function allows the conversion of CAN frame formats in the HW gateway, i.e., converting classical CAN frames to CAN-FD frames or vice versa.

q. Use either <xxx>_FDF_CLASSIC (to convert into classical CAN frames) or <xxx>_FDF_FD (to convert into CAN-FD frames).

r. Use either <xxx>_BRS_SWITCH (to convert into CAN-FD fast data frames, if <xxx>__FDF_FD is set, too) or <xxx>_BRS_NOSWITCH (to use arbitration bit rate only).

s. For (E2) controllers only.
Use either <xxx>_TXBOXMERGE (to merge the lower 6 TX Buffers into 2 CAN-FD buffers with full data length support) or <xxx>_TXBOXNOMERGE (to restrict CAN-FD data size for transmission to 20 bytes).

t. Use either <xxx>_FDONLY (to treat classical CAN frames as errors and don't send them) or <xxx>_FDMIXED (to tolerate both classical CAN and CAN-FD frames for reception and transmission).

u. Use either <xxx>_RXEDGEFILTER_ON (to filter glitches on RX for hard synchronization, which are shorter than 2 dominant TQ), or <xxx>_RXEDGEFILTER_OFF (to allow any recessive to dominant edge for a hard synchronization, if it is within intermission).

v. For (E3) controllers only.

w. On (E2) controller types, the addressing is not indexed. Only one queue is available.
On (E3) controller types, the addressing is indexed (in preparation for several queues to be available in future).

x. Use either <xxx>_TXQ_INT_ONEVERY (every transmission causes an interrupt) or <xxx>_TXQ_INT_ONLAST (interrupt triggered when queue is empty)

y. Use either <xxx>_THL_INT_ONEVERY (every new entry causes an interrupt) or <xxx>_THL_INT_ONLEVEL (interrupt triggered when history list has a fill level of 3/4)

z. Use either <xxx>_THL_ENTRY_ALL (all transmission sources will cause an entry) or <xxx>_THL_ENTRY_QUEUED (only FIFOs and TX Queues will cause entries)

(4) Implementations: (E4)

(4-1) Parameters

UnitNumber_u08: Selected CAN Controller
ChannelNumber_u08: Selected CAN Channel of Controller
Config: Channel configuration structure

(4-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(4-3) Functional Description

The function is named <xxx>_SetChannelConfiguration( ).

Parameters are passed by using referencing a common configuration structure.
The common configuration is setting the bit rate parameters either automatically (if bit rate is specified) or by explicit parameters for the bit timing in time quanta. Also, the sampling point setting has a default value of <*xxx*>_*BT_SPTOPTIMUM*, if no explicit time segment settings are specified.

The channel configuration structure <*xxx*>_*cfg_channel* contains the following elements (when referring to the structure element's names, further information is available in the corresponding user's manual of the product):

Table 5.7 Channel Configuration Structure Elements

Element	Type	Value Range	Description
arb_bitrate [Arbitration Bit Rate]	u32	bits per second 0 ... 1000000	Bit rate to be set for communication arbitration phase in CAN-FD or in general for classic CAN. Set to 0, if automatic bit timing setting is *not* desired.
arb_ samplingpointpos [Sampling Point]	flt	Position in Bit (%/100) 0.5 ... 0.98	Sampling Point to be set for communication arbitration phase in CAN-FD or in general for classic CAN. Set to 0.0, if preferred default setting is acceptable.
data_bitrate [Arbitration Bit Rate]	u32	bits per second 0 ... 20000000 ([a])	Bit rate to be set for communication data phase in CAN-FD. Set to 0, if automatic bit timing setting is *not* desired.
data_ samplingpointpos [Sampling Point]	flt	Position in Bit (%/100) 0.5 ... 0.93	Sampling Point to be set for communication to be set for communication data phase in CAN-FD. Set to 0.0, if preferred default setting is acceptable.
ncfg [Arbitration Bit Timing] ([b])	<xxx>_c_ncfg.nbrp <xxx>_c_ncfg.nsjw <xxx>_c_ncfg.ntseg1 <xxx>_c_ncfg.ntseg2	{0 ... 1023} {0 ... 127} {1 ... 255} {1 ... 127}	Bit Rate Prescaler for Arbitration. Synchronization Jump Width for Arbitration. Time Segment 1 for Arbitration. Time Segment 2 for Arbitration.
dcfg [Data Bit Timing] ([c])	<xxx>_c_dcfg.dbrp <xxx>_c_dcfg.dtseg1 <xxx>_c_dcfg.dtseg2 <xxx>_c_dcfg.dsjw	{0 ... 255} {1 ... 31} {1 ... 15} {0 ... 15}	Bit Rate Prescaler for Data Phase. Time Segment 1 for Data Phase. Time Segment 2 for Data Phase. Synchronization Jump Width for Data Phase.

Table 5.7 Channel Configuration Structure Elements

Element	Type	Value Range	Description
ctr [Channel CTR Register]	<xxx>_c_ctr.chmdc	{0, 1, 2}	Channel state during configuration ([d]). Use either <xxx>_OPMODE_OPER (operation mode), <xxx>_OPMODE_RESET (reset mode) or <xxx>_OPMODE_HALT (halt mode).
	<xxx>_c_ctr.cslpr	-	Not used during channel configuration.
	<xxx>_c_ctr.rtbo	{0, 1}	Force to return from bus-off, if set to 1.
	<xxx>_c_ctr.ie	{one or several flags}	Interrupt sources of channel to be enabled. ([e])
	<xxx>_c_ctr.bom	{0, 1, 2, 3}	Bus off recovery specification (4 options) ([f])
	<xxx>_c_ctr.errd	{0, 1}	First error blocks further reports, if not set.
	<xxx>_c_ctr.ctme	{0, 1}	Test mode enable flag (if set, see *ctms*)
	<xxx>_c_ctr.ctms	{1, 2, 3}	Test mode selection ([g])
	<xxx>_c_ctr.crct	{0, 1}	CRC error test activation, if set
	<xxx>_c_ctr.rom	{0, 1}	Restricted operation mode, if set
fdctr [CAN-FD Control Settings]	<xxx>_c_fdctr.eocclr	{0, 1}	Clear error occurrence counter, if set
	<xxx>_c_fdctr.socclr	{0, 1}	Clear successful occurrence counter, if set
fdcfg [CAN-FD Configuration Settings]	<xxx>_c_fdcfg.eoccfg	{0, 1, 2, 4, 5, 6}	Error occurrence counter configuration ([h])
	<xxx>_c_fdcfg.tdcoc	{0, 1}	Transceiver delay compensation offset config. ([i])
	<xxx>_c_fdcfg.tdce	{0, 1}	Transceiver delay compensation enabled, if set ([j])
	<xxx>_c_fdcfg.esic	{0, 1}	Error state indication configuration ([k])
	<xxx>_c_fdcfg.tdco	{0 ... 255}	Transceiver delay compensation offset in communication clock cycles, rounded to full TQ.
	<xxx>_c_fdcfg.gwen	{0, 1}	Multi-gateway function enabled, if set ([l])
	<xxx>_c_fdcfg.gwfdf	{0, 1}	Multi-gateway function configuration for FDF ([m])
	<xxx>_c_fdcfg.gwbrs	{0, 1}	Multi-gateway function configuration for BRS ([n])
	<xxx>_c_fdcfg.fdoe	{0, 1}	CAN-FD only mode enabled, if set ([o])
	<xxx>_c_fdcfg.refe	{0, 1}	Enable RX edge filter, if set ([p])
	<xxx>_c_fdcfg.cloe	{0, 1}	Classical CAN only mode, if set
tmiec [TX Interrupt Enable flags]	u32	{0, 1} for each TX buffer; bit position corresponds with buffer number	Enable transmit interrupt or transmit abort interrupt for transmit buffer(s), if set.
txq[ ] ([q]) [TX Queue configuration, as an array for queues 0...3]	<xxx>_c_txqcc.qe	{0, 1}	Enable TX Queue, if set.
	<xxx>_c_txqcc.gwe	{0, 1}	Enable gateway mode, if set (only queues 0...2)
	<xxx>_c_txqcc.owe	{0, 1}	Enable overwrite mode, if set
	<xxx>_c_txqcc.ie	{0, 1}	TX Queue interrupt enabled, if set.
	<xxx>_c_txqcc.im	{0, 1}	TX Queue interrupt mode. ([r])
	<xxx>_c_txqcc.dc	{0, 2 ... 15 | 31}	Depth of TX Queue (off, 3 to 16 (E2) or 32 (E3) messages).
	<xxx>_c_txqcc.fie	{0, 1}	TX Queue 0..2 full interrupt enabled, if set
	<xxx>_c_txqcc.ofrxie	{0, 1}	TX Queue 0..2 one frame reception interrupt, if set
	<xxx>_c_txqcc.oftxie	{0, 1}	TX Queue one frame transmit interrupt, if set
thlcc [Transmit History configuration]	<xxx>_c_thlcc.thle	{0, 1}	Enable Transmit History, if set.
	<xxx>_c_thlcc.ie	{0, 1}	Transmit History interrupt enabled, if set.
	<xxx>_c_thlcc.im	{0, 1}	Transmit History interrupt mode ([s]).
	<xxx>_c_thlcc.dte	{0, 1}	Specification what gets an entry in the list ([t]).
	<xxx>_c_thlcc.dge	{0, 1}	Also capture HW Gateway transfers, if set.
cfcc [Multi-Purpose FIFO configuration]	-	-	Not used within <xxx>_Set[...]Configuration( ). See <xxx>_SetCOMFIFOConfiguration( ) for initialization.
cfcce [Multi-Purpose FIFO enhanced configuration]			

a. The maximum value depends on the capabilities of the hardware implementation (clock speed).

b. Values in this register are setting the configuration with one more than programmed. These values are used, if automatic bit timing setting is suppressed by specifying 0 or 0.0 for arbitration bitrate / sampling point.

c. Values in this register are setting the configuration with one more than programmed. These values are used, if automatic bit timing setting is suppressed by specifying 0 or 0.0 for data bitrate / sampling point.

d. The setting is not persistent and used only during the configuration phase. If no special requirement is given, it is recommended to set this to <xxx>_OPMODE_RESET. After configuration, the state will be <xxx>_OPMODE_HALT.

e. Channel interrupt sources are:
<xxx>_CINT_BUSERR, <xxx>_CINT_WARNING, <xxx>_CINT_PASSIVE, <xxx>_CINT_BUSOFF, <xxx>_CINT_RECOVERY, <xxx>_CINT_OVERLOAD, <xxx>_CINT_BUSLOCK, <xxx>_CINT_ARBLOST, <xxx>_CINT_TXABORT, <xxx>_CINT_ERRCOVF, <xxx>_CINT_SUCCOVF, <xxx>_CINT_TDCVIOL.

f. Channel Bus Off Recovery options are:
<xxx>_BOM_ISO (according to ISO), <xxx>_BOM_HALTBOFF (HALT mode before recovery), <xxx>_BOM_HALTRECV (HALT mode after recovery), <xxx>_BOM_SW (recovery control by software, no ISO recovery phase).

g. Set *chmdc* to <xxx>_OPMODE_HALT, when using test modes. Test modes are:
<xxx>_TEST_RXONLY (Receive-Only operation mode), <xxx>_TEST_EXTLOOP (external loop self-test mode - includes transceiver), <xxx>_TEST_INTLOOP (internal loop self-test mode - excludes transceiver).

h. User either <xxx>_EOC_ALLTXRX (all frames), <xxx>_EOC_ALLTX (all transmitter frames), <xxx>_EOC_ALLRX (all receiver frames), <xxx>_EOC_ALLTXRXFD (only CAN-FD frames), <xxx>_EOC_ALLTXFD (only CAN-FD transmitter frames) or <xxx>_EOC_ALLRXFD (only CAN-FD receiver frames)

i. Use either <xxx>_TDC_OFFSETONLY (to disable measured offset) or <xxx>_TDC_MEASOFFSET (to combine measured offset with additional offset added by soft setting).

j. Use either <xxx>_TDC_ENABLE (to enable) or <xxx>_TDC_DISABLE (to disable).

k. Use either <xxx>_ESI_BYNODE (to set ESI by local node status) or <xxx>_ESI_BYACTBUFFER (to set ESI by software within TX buffers, if the node is error active).

l. Use either <xxx>_MULTIGW_ENABLE (activate HW multi-gateway function) or <xxx>_MULTIGW_DISABLE (disable HW multi-gateway function). The multi-gateway function allows the conversion of CAN frame formats in the HW gateway, i.e., converting classical CAN frames to CAN-FD frames or vice versa.

m. Use either <xxx>_FDF_CLASSIC (to convert into classical CAN frames) or <xxx>_FDF_FD (to convert into CAN-FD frames).

n. Use either <xxx>_BRS_SWITCH (to convert into CAN-FD fast data frames, if <xxx>__FDF_FD is set, too) or <xxx>_BRS_NOSWITCH (to use arbitration bit rate only).

o. Use either <xxx>_FDONLY (to treat classical CAN frames as errors and don't send them) or <xxx>_FDMIXED (to tolerate both classical CAN and CAN-FD frames for reception and transmission).

p. Use either <xxx>_RXEDGEFILTER_ON (to filter glitches on RX for hard synchronization, which are shorter than 2 dominant TQ), or <xxx>_RXEDGEFILTER_OFF (to allow any recessive to dominant edge for a hard synchronization, if it is within intermission).

q. Flags GWE, FIE, FRXIE are only applicable for TX Queues 0...2. If a TX Queue is not used in gateway mode (GWE=0), then the flags FIE and FRXIE are ignored.

r. Use either <xxx>_TXQ_INT_ONEVERY (every transmission causes an interrupt) or <xxx>_TXQ_INT_ONLAST (interrupt triggered when queue is empty)

s. Use either <xxx>_THL_INT_ONEVERY (every new entry causes an interrupt) or <xxx>_THL_INT_ONLEVEL (interrupt triggered when history list has a fill level of 3/4)

t. Use either <xxx>_THL_ENTRY_ALL (all transmission sources will cause an entry) or <xxx>_THL_ENTRY_QUEUED (only FIFOs and TX Queues will cause entries)

(5) Implementations: (F)

(5-1) Parameters

UnitNumber_u08: Selected CAN Controller
Config: Channel configuration structure
ErrorStatus_pu08: Error status return by reference

(5-2) Return Values

<*xxx*>_*ERROR* on parameter failures or an error status indication, otherwise <*xxx*>_*OK*.

(5-3) Functional Description

The function is named <xxx>_SetConfiguration( ).

Parameters are passed by using referencing a common configuration structure.
The common configuration is setting the bit rate parameters either automatically (if bit rate is specified) or by explicit parameters for the bit timing in time quanta. Also, the sampling point setting has a default value of <*xxx*>_*BT_SPTOPTIMUM*, if no explicit time segment settings are specified.

The function returns error information by reference, which can be the following:
<xxx>_CONFIG_OK - No error
<xxx>_CONFIG_ERROR_UNITNOTEXIST - Invalid unit number was specified
<xxx>_CONFIG_ERROR_NOTININIT - Must be in INIT mode to allow configuration setting
<xxx>_CONFIG_ERROR_BITTIMING - Bit timing cannot be achieved by given settings
<xxx>_CONFIG_ERROR_OUTOFRAM - Configuration of RAM exceeds the RAM size
<xxx>_CONFIG_ERROR_HWTIMEOUT - No reaction from hardware (timeout limit)

The timeout limit can be set by using the <xxx>_SHUTDOWNTIMEOUT *#define* constant in the driver's mapping definition (see 5.4 for details and file information).

The common configuration structure <*xxx*>_*config* contains the following elements (when referring to the structure element's names, further information is available in the corresponding user's manual of the product):

Table 5.8 Common Configuration Structure Elements

Element	Type	Value Range	Description
arb_bitrate [Arbitration Bit Rate]	u32	bits per second 0 ... 1000000	Bit rate to be set for communication arbitration phase in CAN-FD or in general for classic CAN. Set to 0, if automatic bit timing setting is *not* desired.
arb_ samplingpointpos [Sampling Point]	flt	Position in Bit (%/100) 0.5 ... 0.98	Sampling Point to be set for communication arbitration phase in CAN-FD or in general for classic CAN. Set to 0.0, if preferred default setting is acceptable.
data_bitrate [Arbitration Bit Rate]	u32	bits per second 0 ... 20000000 ([a])	Bit rate to be set for communication data phase in CAN-FD. Set to 0, if automatic bit timing setting is *not* desired.
data_ samplingpointpos [Sampling Point]	flt	Position in Bit (%/100) 0.5 ... 0.93	Sampling Point to be set for communication to be set for communication data phase in CAN-FD. Set to 0.0, if preferred default setting is acceptable.
btp [Arbitration Bit Timing] ([b])	<xxx>_c_btp.sjw <xxx>_c_btp.tseg2 <xxx>_c_btp.tseg1 <xxx>_c_btp.brp	{0 ... 15} {0 ... 15} {1 ... 63} {0 ... 1023}	Synchronization Jump Width for Arbitration. Time Segment 2 for Arbitration. Time Segment 1 for Arbitration. Bit Rate Prescaler for Arbitration.

Table 5.8 Common Configuration Structure Elements

Element	Type	Value Range	Description
fbtp [Data Bit Timing] (c)	<xxx>_c_fbtp.fsjw <xxx>_c_fbtp.ftseg2 <xxx>_c_fbtp.ftseg1 <xxx>_c_fbtp.fbrp <xxx>_c_fbtp.tdc <xxx>_c_fbtp.tdco	{0 ... 3} {0 ... 7} {1 ... 15} {0 ... 31} {0 ... 1} {0 ... 31}	Synchronization Jump Width for Data Phase. Time Segment 2 for Data Phase. Time Segment 1 for Data Phase. Bit Rate Prescaler for Data Phase. Enables Transceiver Delay Compensation if set. Transceiver Delay Compensation value. (d)
tscc [Timestamp Counter Configuration]	<xxx>_c_tscc.tss <xxx>_c_tscc.tcp	{0, 1, 2} {0 ... 15}	Timestamp Selection. (e) Timestamp and Timeout counter prescaler. (f)
gfc [Global Filter Configuration]	<xxx>_c_gfc.rrfe <xxx>_c_gfc.rrfs <xxx>_c_gfc.anfe <xxx>_c_gfc.anfs	{0 ... 1} {0 ... 1} {0, 1, 2} {0, 1, 2}	Extended Remote Frame Filter setting. (g) Standard Remote Frame Filter setting. (h) Extended Frames handling, if no filter matches. (i) Standard Frames handling, if no filter matches. (j)
xidam [Extended ID AND Mask]	<xxx>_c_xidam.eidm	{0 ... 0x1FFFFFFF}	Additional masking of Extended ID for SAE J1939. If a bit is set, the mask is transparent.
rxf*n*c, $n \in$ {0 ... 1} [RX FIFO *n* Configuration]	<xxx>_c_rxfc.fsa <xxx>_c_rxfc.fs <xxx>_c_rxfc.fwm <xxx>_c_rxfc.fom	- - {0 ... 64} {0 ... 1}	This value is calculated automatically by the driver. Not used, set in <xxx>_ramconfig structure. Watermark interrupt level of FIFO (0 to disable). RX FIFO operation mode. (k)
txbc [TX Buffer Configuration]	<xxx>_c_txbc.tbsa <xxx>_c_txbc.ndtb <xxx>_c_txbc.tfqs <xxx>_c_txbc.tfqm	- - - {0 ... 1}	This value is calculated automatically by the driver. Not used, set in <xxx>_ramconfig structure. Not used, set in <xxx>_ramconfig structure. TX FIFO operation mode (l)
txefc [TX Event Buffer Configuration]	<xxx>_c_txefc.efsa <xxx>_c_txefc.efs <xxx>_c_txefc.efwm	- - {0 ... 32}	This value is calculated automatically by the driver. Not used, set in <xxx>_ramconfig structure. Watermark interrupt level (0 to disable).
ramconfig [RAM Configuration]	<xxx>_ramconfig. ... mcan_a_stdfilters_count mcan_a_extfilters_count mcan_a_fifo0_size mcan_a_fifo1_size mcan_a_rxbuffers_count mcan_a_txbuffers_count mcan_a_txqueue_size mcan_a_thl_size	(prefix for all) (m)	 Number of Standard ID Filter objects. Number of Extended ID Filter objects. Number of receive objects for RX FIFO 0. Number of receive objects for RX FIFO 1. Number of standard receive buffers. Number of transmit buffers in total. Number of transmit objects for TX FIFO or queue. Number of entries of the Event Buffer.
rxesc [RX Buffer/FIFO Size Configuration]	<xxx>_c_rxesc.f0ds <xxx>_c_rxesc.f1ds <xxx>_c_rxesc.rbds	{0 ... 7}	Size of data field for CAN-FD operation. 000= 8 bytes 001= 12 bytes 010= 16 bytes 011= 20 bytes 100= 24 bytes 101= 32 bytes 110= 48 bytes 111= 64 bytes
txesc [TX Buffer Size Configuration]	<xxx>_c_txesc.tbds		

a. The maximum value depends on the capabilities of the hardware implementation (clock speed).

b. Values in this register are setting the configuration with one more than programmed. These values are used, if automatic bit timing setting is suppressed by specifying 0 or 0.0 for arbitration bitrate / sampling point.

c. Values in this register are setting the configuration with one more than programmed. These values are used, if automatic bit timing setting is suppressed by specifying 0 or 0.0 for data bitrate / sampling point.

d. Additional offset from measured delay to secondary sample point in counts of communication clock cycles.

e. Use either <xxx>_TSS_OFF (no timestamp), <xxx>_TSS_INTERNAL (use internal counter), or <xxx>_TSS_EXTERNAL (use other counter provided by device implementation)

f. Value is amount of bit times of the configured arbitration bit rate plus one.

g. Use either <xxx>_GFC_REMOTEACCEPT (accept remote frames in general) or <xxx>_GFC_REMOTEREJECT (reject remote frames in general).

h. Use either <xxx>_GFC_REMOTEACCEPT (accept remote frames in general) or <xxx>_GFC_REMOTEREJECT (reject remote frames in general).

i. Use either <xxx>_GFC_NOMATCHFIFO0 (store non-matching frames in RX FIFO 0), <xxx>_GFC_NOMATCHFIFO1 (store non-matching frames in RX FIFO 1) or <xxx>_GFC_NOMATCHREJECT (reject non-matching frames).
j. Use either <xxx>_GFC_NOMATCHFIFO0 (store non-matching frames in RX FIFO 0), <xxx>_GFC_NOMATCHFIFO1 (store non-matching frames in RX FIFO 1) or <xxx>_GFC_NOMATCHREJECT (reject non-matching frames).
k. Use either <xxx>_FIFO_MODE_BLOCKING (to set blocking mode) or <xxx>_FIFO_MODE_OVERWRITE (to set overwrite mode).
l. Use either <xxx>_TXB_FIFOMODE (to set TX buffers as a FIFO) or <xxx>_TXB_QUEUEMODE (to set TX buffers as a transmit queue).
m. The amount of instances for each kind of objects can be chosen in certain ranges, while considering the available RAM size of the M_(TT)CAN controller as specified in the device documentation.

(6) Implementations: (G)

See implementation (F), with some TTCAN extensions (t.b.d.).

5.3.5.4 <xxx>_SetCOMFIFOConfiguration( )

Implementations: (C), (D), (Ex), (E4).

(1) Implementations: (C), (D), (Ex), (E4)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
ChannelNumber_u08:	Selected CAN Channel of Controller
Config:	Channel configuration structure

(1-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(1-3) Functional Description

Parameters are passed by using referencing the same channel configuration structure as used in <xxx>_Set[...]Configuration( ). By <xxx>_SetCOMFIFOConfiguration( ), the part of Multi-Purpose FIFO setup is performed in this separate step. The reason for this approach is, that global Multi-Purpose FIFO setup needs to be done after the RS-CAN(-FD) controller is already initialized in a global operation state.

From the global configuration structure <*xxx*>_*cfg_channel*, the remaining elements are used, as shown in the table below:

Table 5.9 Global Configuration Structure Receive FIFO Initialization Elements

Element	Type	Value Range	Description
cfcc (a) [Multi-Purpose FIFO configuration]	<xxx>_c_cfcc.cfe	-	Not used during FIFO configuration.
	<xxx>_c_cfcc.cfrxie	{0, 1}	FIFO RX Interrupt enable.
	<xxx>_c_cfcc.cftxie	{0, 1}	FIFO TX Interrupt enable.
	<xxx>_c_cfcc.cfpls	{0 ... 7}	Size of FIFO receive objects (b)
	<xxx>_c_cfcc.cfdc	{0 ... 7}	Depth of FIFO (c)
	<xxx>_c_cfcc.cfim	{0, 1}	FIFO Interrupt mode (d)
	<xxx>_c_cfcc.cfigcv	n ∈ {0 ... 7}	Fill level of interrupt generation (see cfim) (e)
	<xxx>_c_cfcc.cfm	{0, 1, 2}	FIFO operation mode (f)
	<xxx>_c_cfcc.cfitss	{0, 1}	Transmission interval timer source selection (g)
	<xxx>_c_cfcc.cfitr	{0, 1}	Transmission interval timer resolution (h)
	<xxx>_c_cfcc.cftml	{0 ... }	Linked Transmit Buffer of FIFO (i)
	<xxx>_c_cfcc.cfitt	{0 ... 255}	Amount of interval timer clocks for FIFO transmission delay.
cfcce (j) [Multi-Purpose FIFO enhanced configuration]	<xxx>_c_cfcce.cffie	{0, 1}	FIFO Full Interrupt enabled, if set
	<xxx>_c_cfcce.cfofrxie	{0, 1}	FIFO one frame reception Interrupt enabled, if set
	<xxx>_c_cfcce.cfoftxie	{0, 1}	FIFO one frame transmit Interrupt enabled, if set
	<xxx>_c_cfcce.cfmowm	{0, 1}	FIFO overwrite mode (k)
	<xxx>_c_cfcce.cfbme	{0, 1}	FIFO buffering disable (temporary stop) (l)

a. Order of elements within structure is different in implementations.

b. Only available in (Ex) controllers.
0: 8 Bytes, 1: 12 Bytes, 2: 16 Bytes, 3: 20 Bytes, 4: 24 Bytes, 5: 32 Bytes, 6: 48 Bytes, 7: 64 Bytes.
Consider maximum amount of available storage when setting this value (see user's manual).

c. 0: 0 Messages (FIFO is disabled), 1: 4 Messages, 2: 8 Messages, 3: 16 Messages, 4: 32 Messages,
5: 48 Messages, 6: 64 Messages, 7: 128 Messages
Consider maximum amount of available storage when setting this value (see user's manual).

d. <xxx>_FIFO_INT_ONLEVEL: use level of rfigcv; <xxx>_FIFO_INT_ONEVERY: on every message

e. Value *n* sets the fill level of generating an interrupt to (*n*+1)/8.
Use constants <xxx>_FIFO_ILEVEL_<*n+1*>D8 to set the level.

f. Use either <xxx>_FIFO_MODE_RX (receive mode), <xxx>_FIFO_MODE_TX (transmit mode) or <xxx>_FIFO_MODE_GW (gateway mode).

g. Use either <xxx>_FIFO_IT_REFCLK (the peripheral clock is used for the timer) or <xxx>_FIFO_IT_BTCLK (the communication bit time quanta clock is used for the timer).

h. Use either <xxx>_FIFO_IT_REFCLK1 (when using the peripheral clock for the timer, this clock is not divided) or <xxx>_FIFO_IT_REFCLK10 (the peripheral clock is divided by 10 as reference for the timer).

i. The range of valid buffer numbers varies among the CAN controller types.

j. Supported by implementation (E4) only.

k. Set either <xxx>_COM_FIFO_OWMODE to allow overwriting of the last received message, if a newly received message arrives while the FIFO is full, or <xxx>_COM_FIFO_DSCMODE, to discard the new received message. Overwriting is only allowed in HW gateway mode.

l. Set this flag during operation to suspend the FIFO transmission, when in FIFO TX mode or HW gateway mode.

5.3.5.5 <xxx>_CreateInterrupt( )

Implementations: (A), (B), (C), (D), (Ex), (F), (G).

(1) Implementations: (A), (B)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
MachineNumber_u08:	Selected CAN Channel of Controller
IntNumber_u08:	Interrupt specification
SetIntLevel_u08:	Interrupt enable/level for interrupt controller setting
FunctionVector:	Pointer to a function returning void as a callback function

(1-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(1-3) Functional Description

For each channel of a CAN controller, each interrupt source can be selected by IntNumber_u08 and their corresponding interrupt can be enabled in the interrupt controller by setting the level control by SetIntLevel_u08. If the interrupt is called, the given callback function vector is executed after processing the essential issues of the CAN driver, for further processing by the user application. If the function vector is *NULL*, then the callback is disabled.

The essential processing of the CAN driver is to clear any pending interrupts and to record any error interrupt status in the variable <*xxx*>_*LastMachineErrorInterrupt_u08[]* array.

The interrupt specification of IntNumber_u08 is as follows:

<xxx>_INT_TX:	Transmit interrupt
<xxx>_INT_RX:	Receive interrupt
<xxx>_INT_ERR:	Error interrupt
<xxx>_INT_WUP:	Wake up interrupt

The interrupt level can be in a range of 0 to 7 (as defined in the applied CPU core systems), using the <*xxx*>_*INTCLEAR #define* constant in the driver's mapping definition (see 5.4 for details and file information) to disable the interrupt.

For each used CAN controller unit and channel, the following dedicated *#define* constants must be set in the driver's mapping definition (see 5.4 for details and file information):

#define <xxx>_INTM<*unit*>C<*channel*>ERR	<*devicefile register name of error interrupt*>
#define <xxx>_INTM<*unit*>C<*channel*>WUP	<*devicefile register name of wake up interrupt*>
#define <xxx>_INTM<*unit*>C<*channel*>RX	<*devicefile register name of receive interrupt*>
#define <xxx>_INTM<*unit*>C<*channel*>TX	<*devicefile register name of transmit interrupt*>

These settings will bind the interrupt controller registers to the CAN controller unit <*unit*>, channel <*channel*>; where each register is represented by its device file name.

(2) Implementations: (C), (D), (Ex), (E4)

(2-1) Parameters

UnitNumber_u08:	Selected CAN Controller
ChannelNumber_u08:	Selected CAN Channel of Controller
IntNumber_u08:	Interrupt specification
SetIntLevel_u16:	Interrupt enable/level for interrupt controller setting
FunctionVector:	Pointer to a function returning void as a callback function

(2-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(2-3) Functional Description

For each channel of a CAN controller, each interrupt source can be selected by IntNumber_u08 and their corresponding interrupt can be enabled in the interrupt controller by setting the level control by SetIntLevel_u16. If the interrupt is called, the given callback function vector is executed after processing the essential issues of the CAN driver, for further processing by the user application. If the function vector is *NULL*, then the callback is disabled.
In order to set global interrupts, the constant <xxx>_GLOBAL must be used as channel number.
The essential processing of the CAN driver is to clear any pending interrupts and to record any error interrupt status in the variable <*xxx*>_*LastErrorCode_Global_u08* variable. Interrupt status is recorded in <*xxx*>_*InterruptFlag_Global_u08* and <*xxx*>_*InterruptFlag_Channel_u08[]* array.

The interrupt specification of IntNumber_u08 is as follows:

<xxx>_INT_GERR:	Global errors of CAN controller
<xxx>_INT_RXF*n*:	Global receive FIFO interrupt (specify *n* = 0 ...7)
<xxx>_INT_TX:	Transmit interrupt
<xxx>_INT_TXA:	Transmit abortion interrupt
<xxx>_INT_TXQ:	Transmit queue interrupt
<xxx>_INT_CERR:	Channel error interrupt
<xxx>_INT_TXHL:	Transmit history list interrupt
<xxx>_INT_RXCF:	Channel multi purpose FIFO receive interrupt
<xxx>_INT_TXCF:	Channel multi purpose FIFO transmit interrupt
<xxx>_INT_GWERR:	Channel gateway error interrupt [(E4) implementation only]

The interrupt level is a value as defined in the applied CPU core systems EI level interrupt control registers, using the <*xxx*>_*INTCLEAR #define* constant in the driver's mapping definition (see 5.4 for details and file information) to disable the interrupt. Use the <*xxx*>_*INTENABLEDEFAULT #define* constant by *OR* combination, in order to always set the *TBxxx* flag of the interrupt controller table reference mode - unless the vector table is mapped differently.

For each used CAN controller unit and channel, the following dedicated *#define* constants must be set in the driver's mapping definition (see 5.4 for details and file information, *n* = 0 ... 7):

#define <xxx>_INT_M<*unit*>GERR	*<devicefile register name of global error interrupt>*
#define <xxx>_INT_M<*unit*>RXF*n*	*<devicefile register name of global RX FIFO interrupt>*
#define <xxx>_INT_M<*unit*>TX<*channel*>	*<devicefile register name of transmit interrupt>*
#define <xxx>_INT_M<*unit*>TXA<*channel*>	*<devicefile register name of transmit abort interrupt>*
#define <xxx>_INT_M<*unit*>TXQ<*channel*>	*<devicefile register name of transmit queue interrupt>*
#define <xxx>_INT_M<*unit*>ERR<*channel*>	*<devicefile register name of channel error interrupt>*
#define <xxx>_INT_M<*unit*>THL<*channel*>	*<devicefile register name of transmit history list interrupt>*
#define <xxx>_INT_M<*unit*>RXCF<*channel*>	*<devicefile register name of multi purpose FIFO RX interrupt>*
#define <xxx>_INT_M<*unit*>TXCF<*channel*>	*<devicefile register name of multi purpose FIFO TX interrupt>*
#define <xxx>_INT_M<*unit*>GWERR<*channel*>	*<devicefile register name of gateway error interrupt> [(E4) implementation only]*

These settings will bind the interrupt controller registers to the CAN controller unit <*unit*>, channel <*channel*>; where each register is represented by its device file name.

(3) Implementations: (F), (G)

(3-1) Parameters

UnitNumber_u08:	Selected CAN Controller
IntLineNumber_u08:	Interrupt line specification
SetIntLevel_u16:	Interrupt enable/level for interrupt controller setting
FunctionVector:	Pointer to a function returning void as a callback function

(3-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(3-3) Functional Description

For each CAN controller, each interrupt line can be selected by IntLineNumber_u08 and their corresponding interrupt can be enabled in the interrupt controller by setting the level control by SetIntLevel_u16. If the interrupt is called, the given callback function vector is executed after processing the essential issues of the CAN driver, for further processing by the user application. If the function vector is *NULL*, then the callback is disabled.

The essential processing of the CAN driver is to clear any pending interrupts and to record the interrupt status in the variables <*xxx*>_*InterruptFlag_Line_u08* and <*xxx*>_*InterruptFlag_Unit_u08* variables.

The interrupt specification of IntNumber_u08 is as follows:

<xxx>_INT_LINE_0:	Interrupt line 0
<xxx>_INT_LINE_1:	Interrupt line 1
<xxx>_INT_LINE_FE:	Interrupt line of debug message reception with filter event line 2

The interrupt level is a value as defined in the applied CPU core systems EI level interrupt control registers, using the <*xxx*>_*INTCLEAR #define* constant in the driver's mapping definition (see 5.4 for details and file information) to disable the interrupt. Use the <*xxx*>_*INTENABLEDEFAULT #define* constant by *OR* combination, in order to always set the *TBxxx* flag of the interrupt controller table reference mode - unless the vector table is mapped differently.

For each used CAN controller unit and channel, the following dedicated *#define* constants must be set in the driver's mapping definition (see 5.4 for details and file information, *n* = 0 ... 7):

#define <xxx>_INT_M<*unit*>LINE0	*<devicefile register name of line 0 interrupt>*
#define <xxx>_INT_M<*unit*>LINE1	*<devicefile register name of line 1 interrupt>*
#define <xxx>_INT_M<*unit*>LINEFE	*<devicefile register name of debug message FE2 interrupt>*

These settings will bind the interrupt controller registers to the CAN controller unit <*unit*>; where each register is represented by its device file name.

5.3.5.6 <xxx>_SetInterrupt( )

Implementations: (A), (B), (C), (D), (Ex), (E4), (F), (G).

(1) Implementations: (A), (B)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
MachineNumber_u08:	Selected CAN Channel of Controller
InterruptMask_u16:	Interrupt mask

(1-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(1-3) Functional Description

The function allows to enable and disable interrupt sources of the CAN controller unit, depending on the addressed CAN controller channel. In order to enable an interrupt, see the user's manual of the device: the value given for InterruptMask_u16 is written into the CAN controller's C*n*IE register, where *n* corresponds with the given value of MachineNumber_u08. Set corresponding bits to enable, clear bits to disable an interrupt[1].

(2) Implementations: (C), (D), (Ex)

(2-1) Parameters

UnitNumber_u08:	Selected CAN Controller
ChannelNumber_u08:	Selected CAN Channel of Controller
InterruptSelection_u08:	Interrupt source selection
InterruptSubSelection_u08:	Interrupt sub-source selection

(2-2) Functional Description

The function allows to enable and disable interrupt sources of the CAN controller unit, depending on the addressed CAN controller channel and the interrupt source selection. Use the following constants to select an interrupt source:

Table 5.10 Interrupt Sources of Implementations (C), (D) and (Ex)

Channel Specification	Interrupt Source (InterruptSelection_u08)	Interrupt Sub-Source (InterruptSubSelection_u08)	Description
<xxx>_GLOBAL	<xxx>_INT_GERR ([a])	<xxx>_GINT_NONE	Disable global error interrupts
		<xxx>_GINT_DLCCHECK	DLC smaller than expected interrupt
		<xxx>_GINT_MSGLOST	Message lost interrupt
		<xxx>_GINT_THLLOST	Transmit history entry lost interrupt
		<xxx>_GINT_RAMPARITY	RAM parity failure interrupt
	<xxx>_INT_RXF*n*	{0, 1}	Enable (1) or disable (0) common receive FIFO *n* interrupt

1. Note that using this function during operation may cause lost interrupts, as the CAN controller does not store interrupt events.

Table 5.10 Interrupt Sources of Implementations (C), (D) and (Ex)

Channel Specification	Interrupt Source (InterruptSelection_u08)	Interrupt Sub-Source (InterruptSubSelection_u08)	Description
Channel *c* (b)	<xxx>_INT_TX	{0 ... *<available TX buffers>*-1}	Set transmit interrupt of buffer
	<xxx>_INT_TXA		Set transmit abort interrupt of buffer
	<xxx>_INT_TXQ	{0, 1}	Enable (1) or disable (0) transmit queue interrupt
	<xxx>_INT_TXHL		Enable (1) or disable (0) transmit history list interrupt
	<xxx>_INT_CERR	See (c)	
	<xxx>_INT_RXCF	{0 ... *<available multi-purpose FIFOs per channel>-1}*	Set receive interrupt of multi-purpose FIFO
	<xxx>_INT_TXCF		Set transmit interrupt of multi-purpose FIFO

a. Use OR operation of sub-sources combination to enable one or several global interrupts.
b. The channel number *c* is given in the range of {0 ,,, *<available channels>*-1}.
When addressing a channel, use <xxx>_INT_NOINT to disable an interrupt (all interrupts of the specified source).
c. Channel error interrupts are one or a combination of the following:

<xxx>_CINT_OFF:	Disable channel error interrupt
<xxx>_CINT_BUSERR:	Bit error channel error interrupt
<xxx>_CINT_WARNING:	Error warning level reached (increasing) channel error interrupt
<xxx>_CINT_PASSIVE:	Error passive level reached (increasing) channel error interrupt
<xxx>_CINT_BUSOFF:	Bus off entered channel error interrupt
<xxx>_CINT_RECOVERY:	Bus off recovery complete channel error interrupt
<xxx>_CINT_OVERLOAD:	Overload frame detected channel error interrupt
<xxx>_CINT_BUSLOCK:	Bus locked dominant channel error interrupt
<xxx>_CINT_ARBLOST:	Arbitration lost channel error interrupt

(3) Implementations: (E4)

(3-1) Parameters

UnitNumber_u08: Selected CAN Controller
ChannelNumber_u08: Selected CAN Channel of Controller
InterruptSelection_u08: Interrupt source selection
InterruptSubSelection_u08: Interrupt sub-source selection
InterruptEnable_u08: Interrupt activation

(3-2) Functional Description

The function allows to enable and disable interrupt sources of the CAN controller unit, depending on the addressed CAN controller channel and the interrupt source selection. Use the following constants to select an interrupt source:

Table 5.11 Interrupt Sources of Implementation (E4)

Channel Specification	Interrupt Source (InterruptSelection_u08)	Interrupt Sub-Source (InterruptSubSelection_u08)	Interrupt Enable (InterruptEnable_u08)	Description
<xxx>_GLOBAL	<xxx>_INT_GERR (a)	(not used)	<xxx>_GINT_NONE	Disable global error interrupts
			<xxx>_GINT_DLCCHECK	DLC smaller than expected
			<xxx>_GINT_MSGLOST	Message lost
			<xxx>_GINT_THLLOST	Transmit history entry lost
			<xxx>_GINT_FDMSGOVF	CAN-FD message overflow
			<xxx>_GINT_GWTXQOVR	TX Queue message overwrite error
			<xxx>_GINT_GWTXQLOST	TX Queue message lost
			<xxx>_GINT_GWFIFOOVR	HW gateway FIFO message overwrite error
	<xxx>_INT_RXF*n*		{0, 1}	Enable (1) or disable (0) common receive FIFO *n* interrupt
Channel *c* (b)	<xxx>_INT_TX	{0 ... <xxx>_MAXTXBUFFERS - 1}	{0, 1}	Clear or set transmit interrupt of selected buffer
	<xxx>_INT_TXA		{0, 1}	Clear or set transmit abort interrupt of selected buffer
	<xxx>_INT_TXQ	{0 ... <xxx>_MAXTXQUEUES - 1}	{0, 1}	Clear or set selected transmit queue interrupt
	<xxx>_INT_GWERR	(not used)	{0, 1}	Clear or set both TX-Queue and multi-purpose FIFO full interrupts (used in gateway mode)
	<xxx>_INT_TXHL		{0, 1}	Clear or set transmit history list interrupt
	<xxx>_INT_CERR		See (c)	Clear or set channel related interrupt
	<xxx>_INT_RXCF	{0 ... <xxx>_MAXCOMFIFOS - 1}	<xxx>_INT_DISABLE <xxx>_INT_ENABLE <xxx>_INT_ENOFRX <xxx>_INT_EN_ALL	multi-purpose FIFO: Disable interrupt Enable interrupt Enable for every frame Enable all interrupts
	<xxx>_INT_TXCF			

a. Use OR operation of sub-sources combination to enable one or several global interrupts.
b. The channel number *c* is given in the range of {0 ,,, <*available channels*>-1}.
When addressing a channel, use <xxx>_INT_NOINT to disable an interrupt (all interrupts of the specified source).

c. Channel error interrupts are one or a combination of the following:

<xxx>_CINT_OFF:	Disable channel error interrupt
<xxx>_CINT_BUSERR:	Bit error channel error interrupt
<xxx>_CINT_WARNING:	Error warning level reached (increasing) channel error interrupt
<xxx>_CINT_PASSIVE:	Error passive level reached (increasing) channel error interrupt
<xxx>_CINT_BUSOFF:	Bus off entered channel error interrupt
<xxx>_CINT_RECOVERY:	Bus off recovery complete channel error interrupt
<xxx>_CINT_OVERLOAD:	Overload frame detected channel error interrupt
<xxx>_CINT_BUSLOCK:	Bus locked dominant channel error interrupt
<xxx>_CINT_ARBLOST:	Arbitration lost channel error interrupt
<xxx>_CINT_TXABORT:	Transmit Abort interrupt
<xxx>_CINT_ERRCOVF:	Error occurrence counter overflow interrupt (see field “fdcfg” of channel configuration)
<xxx>_CINT_SUCCOVF:	Successful occurrence counter overflow interrupt (opposite count of previous)
<xxx>_CINT_TDCVIOL:	Transmit delay compensation violation error interrupt

(4) Implementations: (F)

(4-1) Parameters

UnitNumber_u08:	Selected CAN Controller
InterruptType_u32:	Interrupt source selection
InterruptLineSelection_u08:	Interrupt line specification
InterruptEnable_bit:	Enable / disable flag

(4-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(4-3) Functional Description

The function allows to enable and disable interrupt sources of the CAN controller unit, depending on the interrupt source selection and the selected interrupt line, on which the interrupt is signalized (see <*xxx*>_*CreateInterrupt( )*). Use the following constants to select an interrupt source:

<xxx>_CINT_OFF	No interrupt selected
<xxx>_CINT_ALL	All interrupt sources selected
<xxx>_CINT_RF0N	RX FIFO 0 new message arrived
<xxx>_CINT_RF0W	RX FIFO 0 watermark fill level reached
<xxx>_CINT_RF0F	RX FIFO 0 full
<xxx>_CINT_RF0L	RX FIFO 0 message lost
<xxx>_CINT_RF1N	RX FIFO 1 new message arrived
<xxx>_CINT_RF1W	RX FIFO 1 watermark fill level reached
<xxx>_CINT_RF1F	RX FIFO 1 full
<xxx>_CINT_RF1L	RX FIFO 1 message lost
<xxx>_CINT_DRX	RX Buffer new message arrived
<xxx>_CINT_HPM	High priority message arrived
<xxx>_CINT_TC	Transmission completion
<xxx>_CINT_TCF	Transmission cancellation completed (aborted successfully)
<xxx>_CINT_TFE	TX FIFO empty
<xxx>_CINT_TEFN	Transmit history list new message added
<xxx>_CINT_TEFW	Transmit history list watermark fill level reached
<xxx>_CINT_TEFF	Transmit history list full
<xxx>_CINT_TEFL	Transmit history list entry lost
<xxx>_CINT_ELO	Error logging overflow
<xxx>_CINT_EP	Error passive state entered
<xxx>_CINT_EW	Error warning state entered
<xxx>_CINT_BO	Bus off state entered
<xxx>_CINT_CRCE	CRC error detected
<xxx>_CINT_BE	Bit error detected
<xxx>_CINT_ACKE	Acknowledge error detected
<xxx>_CINT_FOE	Format error detected
<xxx>_CINT_STE	Stuff error detected
<xxx>_CINT_TSW	Timestamp counter wrap around occurred
<xxx>_CINT_MRAF	Message RAM access failure detected
<xxx>_CINT_TOO	Timeout occurred of RX FIFO 0/1 or transmit history list
<xxx>_CINT_BEC	Message RAM bit error detected and corrected by ECC
<xxx>_CINT_BEU	Message RAM bit error detected but not corrected
<xxx>_CINT_WDI	Watchdog interrupt

(5) Implementations: (G)

See implementation (F), with additional TTCAN interrupt sources (t.b.d.).

5.3.6 Reception / Filter Configuration

5.3.6.1 <xxx>_SetStdFilterEntry( )

Implementations: (F), (G).

(1) Implementations: (F), (G)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
RuleNumber_u16:	Reception rule number within standard reception filters RAM section
FilterEntry:	Reception rule specification structure

(1-2) Return Values

<*xxx*>*_ERROR* on parameter failures, otherwise <*xxx*>*_OK*.

(1-3) Functional Description

The function is used to create a new CAN standard ID reception rule, which specifies what kind of arrived standard ID messages are treated in which way (whether to store or not, where to store, etc).

Reception rules are checked on each message reception, in the sequence of the rule numbers, which are specified by RuleNumber_u16 - so this is a kind of priority setting. It is not allowed to create gaps between rules. If a rule shall be deleted, it can either be replaced by a new one (allowed to do this on either rule position), or to be replaced by a *disabled* one (only allowed to do as the last rule with highest order number).

The standard ID reception rule (filter specification) is passed by a reception rule structure, <*xxx*>*_filter_std*, whose elements are shown in the table below:

Table 5.12 Standard Filter Initialization Elements

Element	Type	Value Range	Description
sfid2 [Standard Filter ID 2]	u32	{0 ... 0x3FF}	If *sfec* is set to <xxx>_FILTER_STORE_BUFFER: Set this value to <xxx>_FILTER_BUFFER_STORE and add the buffer number to it. Otherwise, the value represents an 11-bit value, depending on *sfec* setting.
sfid1 [Standard Filter ID 1]	u32	{0 ... 0x3FF}	Standard ID to match with the reception filter.
sfec [Standard Filter Element Configuration]	u32	<xxx>_FILTER_DISABLED <xxx>_FILTER_STORE_FIFO0 <xxx>_FILTER_STORE_FIFO1 <xxx>_FILTER_REJECT <xxx>_FILTER_PRIORITY <xxx>_FILTER_STORE_BUFFER	Filter is disabled. Message to be stored in RX FIFO 0. (*) Message to be stored in RX FIFO 1. (**) Matching messages will not be stored. Message is a high priority message. (a) Message to be stored in a RX Buffer (see *sfid2*).
sft [Standard Filter Type]	u32	<xxx>_FILTERTYPE_RANGE <xxx>_FILTERTYPE_DUAL <xxx>_FILTERTYPE_CLASSIC	Filter matches within the range of *sfid1 ... sfid2*. Filter matches for *sfid1* and *sfid2*. Filter matches for *sfid1* with *sfid2* as binary mask. (b)

a. Combine this setting with (*) or (**) by OR operation. The high priority interrupt can be triggered upon reception.
b. A cleared bit masks the corresponding bit of the ID, so that this bit will be ignored upon match checking.

5.3.6.2 <xxx>_SetExtFilterEntry( )

Implementations: (F), (G).

(1) Implementations: (F), (G)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
RuleNumber_u16:	Reception rule number within standard reception filters RAM section
FilterEntry:	Reception rule specification structure

(1-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(1-3) Functional Description

The function is used to create a new CAN extended ID reception rule, which specifies what kind of arrived extended ID messages are treated in which way (whether to store or not, where to store, etc).

Reception rules are checked on each message reception, in the sequence of the rule numbers, which are specified by RuleNumber_u16 - so this is a kind of priority setting. It is not allowed to create gaps between rules. If a rule shall be deleted, it can either be replaced by a new one (allowed to do this on either rule position), or to be replaced by a *disabled* one (only allowed to do as the last rule with highest order number).

The extended ID reception rule (filter specification) is passed by a reception rule structure, *<xxx>_filter_ext*, whose elements are shown in the table below:

Table 5.13 Extended Filter Initialization Elements

Element	Type	Value Range	Description
f0 [Extended Filter lower word]	<xxx>_filter_f0.efid1 <xxx>_filter_f0.efec	{0 ... 0x1FFFFFFF} See *sfec* in 5.3.6.1.	Extended ID to match with the reception filter. Extended filter element configuration.
f1 [Extended Filter upper word]	<xxx>_filter_f1.efid2 <xxx>_filter_f1.eft	{0 ... 0x1FFFFFFF} See *sft* in 5.3.6.1.	See *sfid2* in 5.3.6.1. Extended filter type.

5.3.6.3 <xxx>_SetAFLEntry( )

Implementations: (C), (D), (Ex), (E4).

(1) Implementations: (C), (D), (Ex)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
ChannelNumber_u08:	Selected CAN Channel of Controller
RuleNumber_u16:	Reception rule number within standard reception filters RAM section
AFLEntry:	Reception rule specification structure

(1-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(1-3) Functional Description

The function is used to create a new CAN reception rule, which specifies what kind of arrived messages are treated in which way (whether to store or not, where to store, etc).

Reception rules are checked on each message reception, in the sequence of the rule numbers, which are specified by RuleNumber_u16 - so this is a kind of priority setting. It is not allowed to create gaps between rules. If a rule shall be deleted, it can either be replaced by a new one (allowed to do this on either rule position), or to be replaced by an empty one (only allowed to do as the last rule with highest order number).
Within the global configuration, a maximum limit is defined for reception rules per channel. The function is checking that this limit is not crossed, as this could cause a corrupted memory image.

The reception rule (filter specification) is passed by a reception rule structure, <*xxx*>_*a_afl*, whose elements are shown in the table below:

Table 5.14 Filter Rule Initialization Elements

Element	Type	Value Range	Description
id [Identifier specification]	<xxx>_a_aflid.id <xxx>_a_aflid.lb <xxx>_a_aflid.rtr <xxx>_a_aflid.ide	{0 ... 0x1FFFFFFF} {0 , 1} {0 , 1} {0 , 1}	Identifier of message(s) to be receivable. Rule applied to own sent messages, if set. (a) Data or remote frame specification. (b) Standard- or extended frame specification (c)
mask [Mask specification]	<xxx>_r_mask.id <xxx>_r_mask.rtr <xxx>_r_mask.ide	{0 ... 0x1FFFFFFF} {0 , 1} {0 , 1}	Mask of identifier for received messages. (d) Data or remote frame specification. (e) Standard- or extended frame specification (f)
ptr0 [Reception target lower word]	<xxx>_a_aflptr0.rmdp <xxx>_a_aflptr0.rmv <xxx>_a_aflptr0.ptr <xxx>_a_aflptr0.dlc	{0 ... *b*} {0 , 1} {0 ... } {0 ... 15}	Reception target buffer number. (g) Reception target buffer activation. (h) Reception label value (rule label, HRH) (i) Minimum DLC of a frame to be received. (j)
ptr1 (k) [Reception target upper word]	<xxx>_a_aflptr1.rxfifomask <xxx>_a_aflptr1.comfifomask	{0 ... 11111111B} {0 ... 1111...1111B}	One flag per RX FIFO to set reception. One flag per multi-purpose FIFO to set reception (depends on product).

a. Own sent messages can be received again by defining rules with the *lb* flag set (<xxx>_AFL_TXENTRY).
This is only allowed in mirror and self test modes, however.
Regular receive rules must be defined with this flag cleared to <xxx>_AFL_RXENTRY.
b. Use either <xxx>_FRAME_DATA (for data frames) or <xxx>_FRAME_REMOTE (for remote frames).
c. Use either <xxx>_ID_STD (for standard frames) or <xxx>_ID_EXT (for extended frames).
d. A cleared bit masks the corresponding bit of the ID, so that this bit will be ignored upon match checking.
Use <xxx>_MASK_IDDONTCARE to create a *basic CAN* reception, and <xxx>_MASK_IDFULLCAN for *full CAN*.
e. Use either <xxx>_FRAME_DATA (for data frames) or <xxx>_FRAME_REMOTE (for remote frames).
f. Use either <xxx>_ID_STD (for standard frames) or <xxx>_ID_EXT (for extended frames).

g. The range of valid receive buffer numbers varies among the CAN controller types.
h. Set the flag to activate reception in the buffer indicated by *rmdp*.
i. For (C), (D) and (E2) controller types, the maximum label value is 0xFFF.
   For (E3) controller types, the maximum label value is 0xFFFF.
j. Set to zero in order to disable the DLC check.
k. The maximum allowed reception targets to be activated at a time in a rule varies among the CAN controller types.
   This maximum includes the standard receive buffer, too.

(2) Implementations: (E4)

(2-1) Parameters

UnitNumber_u08:	Selected CAN Controller
ChannelNumber_u08:	Selected CAN Channel of Controller
RuleNumber_u16:	Reception rule number within standard reception filters RAM section
AFLEntry:	Reception rule specification structure

(2-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(2-3) Functional Description

The function is used to create a new CAN reception rule, which specifies what kind of arrived messages are treated in which way (whether to store or not, where to store, etc).

Reception rules are checked on each message reception, in the sequence of the rule numbers, which are specified by RuleNumber_u16 - so this is a kind of priority setting. It is not allowed to create gaps between rules. If a rule shall be deleted, it can either be replaced by a new one (allowed to do this on either rule position), or to be replaced by an empty one (only allowed to do as the last rule with highest order number).
Within the global configuration, a maximum limit is defined for reception rules per channel. The function is checking that this limit is not crossed, as this could cause a corrupted memory image.

The reception rule (filter specification) is passed by a reception rule structure, <*xxx*>_*a_afl*, whose elements are shown in the table below:

Table 5.15 Filter Rule Initialization Elements

Element	Type	Value Range	Description
id [Identifier specification]	<xxx>_a_aflid.id <xxx>_a_aflid.lb <xxx>_a_aflid.rtr <xxx>_a_aflid.ide	{0 ... 0x1FFFFFFF} {0 , 1} {0 , 1} {0 , 1}	Identifier of message(s) to be receivable. Rule applied to own sent messages, if set. (a) Data or remote frame specification. (b) Standard- or extended frame specification (c)
mask [Mask specification]	<xxx>_a_mask.id <xxx>_a_mask.ifl1 <xxx>_a_mask.rtr <xxx>_a_mask.ide	{0 ... 0x1FFFFFFF} {0 , 1} {0 , 1} {0 , 1}	Mask of identifier for received messages. (d) Upper bit of additional rule label “IFL” Data or remote frame specification. (e) Standard- or extended frame specification (f)
ptr0 [Reception target lower word]	<xxx>_a_aflptr0.dlc <xxx>_a_aflptr0.srd0 <xxx>_a_aflptr0.srd1 <xxx>_a_aflptr0.srd2 <xxx>_a_aflptr0.ifl0 <xxx>_a_aflptr0.rmdp <xxx>_a_aflptr0.rmv <xxx>_a_aflptr0.ptr	{0 ... 15} {0 , 1} {0 , 1} {0 , 1} {0 , 1} {0 ... *b*} {0 , 1} {0 ... 0xFFFF}	Minimum DLC of a frame to be received. (g) Routing target redirection option (h) -- see above -- -- see above -- Lower bit of additional rule label “IFL” Reception target buffer number. (i) Reception target buffer activation. (j) Reception label value (rule label, HRH)
ptr1 (k) [Reception target upper word]	<xxx>_a_aflptr1.rxfifomask <xxx>_a_aflptr1.comfifomask	{0 ... 11111111B} {0 ... 1111...1111B}	One flag per RX FIFO to set reception. One flag per multi-purpose FIFO to set reception (depends on product).

a. Own sent messages can be received again by defining rules with the *lb* flag set (<xxx>_AFL_TXENTRY). This is only allowed in mirror and self test modes, however. Regular receive rules must be defined with this flag cleared to <xxx>_AFL_RXENTRY.
b. Use either <xxx>_FRAME_DATA (for data frames) or <xxx>_FRAME_REMOTE (for remote frames).
c. Use either <xxx>_ID_STD (for standard frames) or <xxx>_ID_EXT (for extended frames).
d. A cleared bit masks the corresponding bit of the ID, so that this bit will be ignored upon match checking. Use <xxx>_MASK_IDDONTCARE to create a *basic CAN* reception, and <xxx>_MASK_IDFULLCAN for *full CAN*.
e. Use either <xxx>_FRAME_DATA (for data frames) or <xxx>_FRAME_REMOTE (for remote frames).
f. Use either <xxx>_ID_STD (for standard frames) or <xxx>_ID_EXT (for extended frames).

g. Set to zero in order to disable the DLC check.
h. If the SRDx flag is set, then the rule redirects a reception from multi-purpose FIFO #x to TX-Queue #x, if the reception target “comfifomask” is set to receive by a multi-purpose FIFO. Only allowed for TX-Queues / FIFOs 0 .. 2.
i. The range of valid receive buffer numbers varies among the CAN controller types.
j. Set the flag to activate reception in the buffer indicated by *rmdp*.
k. The maximum allowed reception targets to be activated at a time in a rule varies among the CAN controller types. This maximum includes the standard receive buffer, too.

5.3.6.4 <xxx>_SetMachineMask( )

Implementations: (A), (B).

(1) Implementations: (A), (B)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
MachineNumber_u08:	Selected CAN Channel of Controller
MaskNumber_u08:	Reception mask number to be altered
MaskExtendedFlag_u16:	Frame type of mask (standard or extended)
MaskValue_u32:	Mask value to be set

(1-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(1-3) Functional Description

Set the global reception mask selected by its number to the given *MaskValue_u32*. Select the frame format by specifying either *<xxx>_ID_STD* (standard frames) or *<xxx>_ID_EXT* (extended frames) as *MaskExtendedFlag_u16*. If a bit in *MaskValue_u32* is set, the corresponding bit in ID fields of message buffers assigned to this mask will be ignored when checking the received ID for matching.

The mask number can be one of *<xxx>_MASK_m*, where *m* is in {0...3} for (A) implementations and in {0...7} for (B) implementations.

5.3.6.5 <xxx>_SetReceiveMessage( )

Implementations: (A), (B).

(1) Implementations: (A), (B)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
MachineNumber_u08:	Selected CAN Channel of Controller
BufferNumber_u08:	Selected message buffer used for reception
ExtendedFrame_u08:	Frame format specification (standard or extended)
RemoteFrame_u08:	Frame type specification (data or remote frame)
InterruptEnable_u08:	Receive interrupt activation for selected message buffer
OverWriteEnable_u08:	Receive message buffer overwriting configuration
BufferMaskFlag_u08:	Selected global reception mask (see *<xxx>_SetMachineMask( )*)
Identifier_u32:	Identifier of message(s) to be received

(1-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(1-3) Functional Description

The function initializes a message buffer of the selected CAN controller unit and channel for usage as a receive buffer.
When setting *ExtendedFrame_u08* to *<xxx>_ID_STD*, the buffer is initialized for standard frames reception; if set to *<xxx>_ID_EXT*, the buffer is initialized for extended frames reception.
When setting *RemoteFrame_u08* > 0, the buffer is initialized for remote frame reception.
When setting *InterruptEnable_u08* > 0, the buffer is initialized to generate receive interrupts.
When setting *OverWriteEnable_u08* > 0, the buffer is allowed to be overwritten by new messages, even if its content was not yet read by using the *<xxx>_ReceiveMessage( )* function.
With the *BufferMaskFlag_u08*, the receive mask is specified, which shall be associated with the buffer. Use the constants *<xxx>_MASK_n* to select the mask to be applied [implementation (A): n = {0...3}, implementation (B): n = {0...7}]. Set *BufferMaskFlag_u08* to zero, if no mask has to be associated (i.e., for defining a *Full-CAN* buffer).
The parameter *Identifier_u32* sets the identifier of the message(s) to be received in the buffer. If a mask is applied, the identifier is set to *don't care* bitwise, at those bits where the mask has a bit set. See *<xxx>_SetMachineMask( )* for details.

5.3.7 Operation and Status

5.3.7.1 <xxx>_Reset( )

Implementations: (A), (B)

(1) Implementations: (A), (B)

(1-1) Parameters

UnitNumber_u08: Selected CAN Controller

(1-2) Return Values

<xxx>_ERROR on parameter failures or hardware timeout, otherwise *<xxx>_OK*.

(1-3) Functional Description

Performs a shut down of the selected CAN controller unit, including all its channels.

In order to shut down a CAN channel synchronously (i.e., not violating the CAN protocol), the function *<xxx>_Start( )* with an operation mode of *<xxx>_OPMODE_INIT* should be used before calling this function.

For implementations of (B), any ongoing soft reset is checked; if this does not end within the *<xxx>_SHUTDOWNTIMEOUT* time, the function stops and returns an error.

The function clears all interrupt sources and stops communication immediately. If this procedure should cause a blocking beyond the *<xxx>_SHUTDOWNTIMEOUT* time, then a forced shut down is initiated. Typically, if the CAN bus is blocked dominant, such a situation may arise.

All message buffers are cleared - for implementation (B), this is done by using the soft reset hardware support. For this reason, it is recommended to use this function after every hard reset, in order to have all CAN memory properly initialized.

(2) Implementations (C), (D), (Ex), (E4)

See *<xxx>_Start( )* function with parameter *<xxx>_OPMODE_RESET*.

(3) Implementations (F), (G)

The reset functionality is implicit with the configuration. See *<xxx>_SetConfiguration( )*.

5.3.7.2 <xxx>_Start( )

Implementations: (A), (B), (C), (D), (Ex), (E4), (F) (G).

(1) Implementations: (A), (B), (C), (D), (Ex), (E4), (F) (G)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
ChannelNumber_u08:	Selected CAN Controller channel [(C), (D), (Ex)]
MachineNumber_u08:	Selected CAN Controller channel [(A), (B)]
OperationMode_u08:	Selected Operation Mode
ClearErrorCounter_u08:	Option to clear *REC* and *TEC* error counters [(A), (B), (C), (D), (Ex), (E4)]
TimeStampSetting_u16:	Sets timestamp counter on defined value [(A), (B), (C), (D), (Ex), (E4)]
TimeOutSetting_u16:	Sets timeout of watchdog [(F), (G)]
ABTDelay_u08:	Message interval in *ABT* operation mode [(A), (B)]

(1-2) Return Values

<xxx>_ERROR on parameter failures or hardware timeout, otherwise *<xxx>_OK*.

(1-3) Functional Description

Activates communication in a dedicated operation mode on the selected CAN controller unit and channel. The selected channel must be in a configured state and the CAN controller memory must be initialized correctly as a precondition to call this function.

The CAN error counters (as specified in *ISO 11898-1*) can be cleared forcibly in some implementations by setting a value > 0 to this parameter.

With the *TimeStampSetting_u16* parameter, in implementations (A) and (B) the timestamp counter can be set on a dedicated value, which in implementations (C), (D), (Ex) and (E4), the timestamp counter can simply be cleared by setting a value > 0.

Using the *TimeOutSetting_u16* parameter in implementations (F) and (G), the watchdog timer prescaler can be set to the given value.

When using the *ABT* mode in implementations (A) or (B), the delay between consecutive messages of an *ABT block* can be set in units of CAN bit times given as $2^{(5+ABTDelay\_u08)}$.

If the parameter *ChannelNumber_u08* is set to *<xxx>_GLOBAL*, then the function is executed for all available channels of the addressed CAN controller unit *UnitNumber_u08*.

The following operation modes can be selected in the dedicated implementations by using the shown codes for *OperationMode_u08*:

Table 5.16 Operation Modes of CAN Controllers

Code	Operation Mode	Implementations	Description
<xxx>_OPMODE_RESET	Reset mode	(C), (D), (Ex), (E4)	Soft reset setting which corresponds with *<xxx>_Reset( )*. Allows configuration.
<xxx>_OPMODE_HALT	Stopped mode	(C), (D), (Ex), (E4)	Stops communication.
<xxx>_OPMODE_INIT	Initialization mode	(A), (B), (F), (G)	Stops communication and allows configuration. (a)

Table 5.16 Operation Modes of CAN Controllers

Code	Operation Mode	Implementations	Description
<xxx>_OPMODE_OPER ([b])	Normal operation modes	(A), (B), (C), (D), (Ex), (E4)	Unrestricted, regular operation.
<xxx>_OPMODE_CLASSIC		(F), (G) ([c])	Regular operation with classical CAN only ([d]).
<xxx>_OPMODE_CANFD			Regular operation with CAN-FD, but no bit rate switching ([e]).
<xxx>_OPMODE_CANFDBRS			Regular operation with CAN-FD, including bit rate switching.
<xxx>_OPMODE_ABT	ABT mode	(A), (B)	Like normal operation mode, but including block transmissions.
<xxx>_OPMODE_RECONLY	Receive-only mode	(A), (B), (C), (D), (Ex), (E4)	Restricted operation with no transmission, no error flagging, no acknowledging to other nodes.
<xxx>_OPMODE_SSHOT	Single-shot mode	(A), (B)	Like normal operation mode, but no retransmission on errors or arbitration loss.
<xxx>_OPMODE_STEST	Internal self-test mode	(A), (B), (C), (D), (Ex), (E4)	Receives own sent messages, no communication to external.

a. In implementations (F) and (G), this command has no effect, if *TimeOutSetting_u16* is zero. If *TimeOutSetting_u16* is not zero, then the *initialization mode* is entered. Use the *<xxx>_Stop( )* API with *<xxx>_PSMODE_INIT* in implementations (F) and (G) to set *initialization mode* without modifying the *TimeOutSetting_u16*.

b. If this mode is *OR*ed with the mode flag <xxx>_OPMODE_RECOVERY in implementations (A) and (B), the function waits until the bus-off recovery of the CAN controller is completed, before returning.
The CAN-FD or classical CAN operation mode selection of implementations (Ex) and (E4) is performed by configuration (as exclusive setting) or per transmission request (see function *<xxx>_SendMessage( )*).

c. The operational modes of implementations (F) and (G) can be combined by *OR*in the mode code with the following options:
<xxx>_SPMODE_RECONLY - corresponds with the receive-only mode <xxx>_OPMODE_RECONLY.
<xxx>_SPMODE_STEST - corresponds with the internal self-test mode <xxx>_OPMODE_STEST.
<xxx>_SPMODE_SSHOT - corresponds with the single shot mode <xxx>_OPMODE_SSHOT.
<xxx>_SPMODE_RESTRICT - restricted operation mode with no transmissions and error flagging, but acknowledging.

d. Classical only mode is also available on (E3) and (E4) controllers, but has to be set by configuration, not operation mode. See *<xxx>_Set[...]Configuration( )*.

e. CAN-FD only mode is also available on (Ex) and (E4) controllers, but has to be set by configuration, not operation mode. See *<xxx>_Set[...]Configuration( )*.

5.3.7.3 <xxx>_Stop( )

Implementations: (A), (B), (C), (D), (Ex), (E4), (F), (G).

(1) Implementations: (A), (B), (C), (D), (Ex), (E4), (F), (G)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
ChannelNumber_u08:	Selected CAN Controller channel [(C), (D), (Ex), (E4)]
MachineNumber_u08:	Selected CAN Controller channel [(A), (B)]
StopMode_u08:	Selected Operation Mode

(1-2) Return Values

<xxx>_ERROR on parameter failures or hardware timeout, otherwise *<xxx>_OK*.

(1-3) Functional Description

Deactivates communication and optionally enters a power saving mode of the selected CAN controller unit and channel. The selected channel must be in a configured state and the CAN controller memory must be initialized correctly as a precondition to call this function.

The following stop / power down modes can be selected in the dedicated implementations by using the shown codes for *StopMode_u08*:

Table 5.17 Stop / Power Down Modes of CAN Controllers

Code	Operation Mode	Implementations	Description
<xxx>_STOPMODE_SLEEP	Sleep mode	(A), (B)	Enters sleep mode. ([a])
<xxx>_STOPMODE_STOP	Stop mode	(A), (B)	Enters stop mode, which disables the CAN controller internal clock and does not wake up automatically. ([b])
<xxx>_OPMODE_SLEEP	Stop mode	(C), (D), (Ex), (E4)	Disables CAN controller clock and freezes its state. ([c],[d])
<xxx>_OPMODE_HALT <xxx>_OPMODE_OPER <xxx>_OPMODE_RECONLY <xxx>_OPMODE_STEST	See *Table 5.16 Operation Modes of CAN Controllers*		Alternative way to enter operational or halted mode as shown in the referenced table.
<xxx>_PSMODE_RUN	Keep operation mode as set in *Table 5.16 Operation Modes of CAN Controllers*	(F), (G)	Has no effect and returns no error.
<xxx>_PSMODE_INIT	Initialization mode		Stops communication synchronously.
<xxx>_PSMODE_STOP	Stop mode		Stops communication synchronously and in addition disables the CAN controller internal clock. Does not wake up from this mode automatically.

a. Sleep mode wakes up on any recessive to dominant edge on the CAN bus and subsequently enters to the lastly selected operation mode. Special care has to be taken when using this mode in combination with the *wakeup* interrupt, as this interrupt occurs only *once*.
The sleep mode is entered synchronously, when the CAN bus is idle and no pending transmissions are there.

b. It is required to specify the *<xxx>_STOPMODE_SLEEP* beforehand, which must have been executed successfully and no wakeup condition have occurred meanwhile. For this reason, this stopping of the CAN controller clock is a synchronous operation.

c. It is recommended to use this mode only after setting the <xxx>_OPMODE_HALT beforehand, because the command is asynchronous to the CAN bus. These CAN controller implementations do not have an automatic wake up functionality implemented. Software must check by appropriate port settings for wake up conditions (i.e., some port edge interrupt).
d. By specifying <xxx>_GLOBAL as *ChannelNumber_u08*, the mode is applied globally, i.e., for all channels belonging to this CAN controller.

5.3.7.4 <xxx>_GetStatus( )

Implementations: (A), (B), (C), (D), (Ex), (E4), (F), (G).

(1) Implementations: (A), (B), (C), (D), (Ex), (E4), (F), (G)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
ChannelNumber_u08:	Selected CAN Controller channel [(C), (D), (Ex), (E4)]
MachineNumber_u08:	Selected CAN Controller channel [(A), (B)]
StatusNumber_u08:	Selected status to be reported
StatusValue_pu08:	Status value returned by reference [(A), (B), (C), (D), (E2)]
StatusValue_pu16:	Status value returned by reference [(E3), (F), (G)]
StatusValue_pu32:	Status value returned by reference [(E4)]

(1-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(1-3) Functional Description

By writing to the referenced pointer address of *StatusValue_pu08 / StatusValue_pu16 / StatusValue_pu32*, the function returns status information of the selected CAN controller unit and channel.

Allowed status selectors by setting *StatusNumber_u08* are depending on the CAN controller implementation and are given as follows:

Table 5.18 Status Return Values of CAN Controllers

Code	Return Value	Implementations	Description
<xxx>_STATUS_OPMODE	Current operation mode	(A), (B), (C), (D), (Ex), (E4), (F), (G) (a)	Use constants as of setting the mode to determine the returned value. See *Table 5.16 Operation Modes of CAN Controllers* and *Table 5.17 Stop / Power Down Modes of CAN Controllers*.
<xxx>_STATUS_PSMODE	Current power save mode		
<xxx>_STATUS_SPMODE	Current operation mode options	(F), (G)	
<xxx>_STATUS_RECEIVE	Current activity	(A), (B), (C), (D), (Ex), (E4), (F), (G)	Set during receive activity.
<xxx>_STATUS_TRANSMIT			Set during transmit activity.
<xxx>_STATUS_BUSOFF	Bus off state		Set if bus off.
<xxx>_STATUS_RXERRCNTLEV	Error counter range state	(A), (B)	0: Less than warning level 1: Warning level 3: Error passive or bus off level
<xxx>_STATUS_TXERRCNTLEV			
<xxx>_STATUS_ERRCNTLEV		(C), (D), (Ex), (E4), (F), (G)	Status value is error level (b)
<xxx>_STATUS_ERRPLEV	Error passive state		Status value set, if error passive
<xxx>_STATUS_RXERRPLEV		(A), (B)	
<xxx>_STATUS_ERRCODE_CLAR	Last error code of arbitration phase / classical CAN mode	(F), (G)	Status value is set to error detail information (c).
<xxx>_STATUS_ERRCODE_FDAT	Last error code of CAN-FD data phase		
<xxx>_STATUS_ROVF	Receive history list status	(A), (B)	Status value is set on overflow.
<xxx>_STATUS_RHPM			Status value is set on empty state.
<xxx>_STATUS_RGPT			Status value returns the history list entry (box number).

Table 5.18 Status Return Values of CAN Controllers

Code	Return Value	Implementations	Description
<xxx>_STATUS_TOVF	Transmit history list / event list status	(A), (B), (C), (D), (Ex), (E4), (F), (G)	Status value is set on overflow.
<xxx>_STATUS_THPM			Status value is set on empty state.
<xxx>_STATUS_TGPT			Status value returns the history list entry label (or box number) ([d]).
<xxx>_STATUS_VALID	Communication status		Status value is set, if the bus integration phase is complete.
<xxx>_STATUS_LASTRECEIVE	Indicate last or currently used buffer in communication.	(A), (B)	Status value indicates last receive buffer.
<xxx>_STATUS_CURTRANSMIT			Status value indicates last transmit buffer.
<xxx>_STATUS_TRERRCOUNT	Transmit error counter	(A), (B), (C), (D), (Ex), (E4), (F), (G)	Status value is TEC
<xxx>_STATUS_RXERRCOUNT	Receive error counter		Status value is REC
<xxx>_STATUS_NEWRXHISTORY	Prepares next history / event list element to be read out (no return value)	(A), (B)	Prepares RX history list
<xxx>_STATUS_NEWTXHISTORY		(A), (B), (C), (D), (Ex), (E4), (F), (G)	Prepares TX history list / event list
<xxx>_STATUS_TXTS	Transmit timestamp	(E3), (F), (G)	Status value is timestamp taken after TX completion.
<xxx>_STATUS_INT_RXFIFO	RX FIFO interrupt status	(C), (D), (Ex), (E4)	Status value is set on interrupt ([e]).
<xxx>_STATUS_INTERRUPT	Interrupt flag of driver		Status value is interrupt flag ([f]).
<xxx>_STATUS_INTS	Pending interrupt status	(A), (B)	Status value is INTS register.
<xxx>_STATUS_MLT_RXFIFO	Message loss indication of dedicated reception target. Status value is set on message loss on...	(C), (Ex), (E4)	... any RX FIFO
<xxx>_STATUS_MLT_COMFIFO			... any multi-purpose FIFO (RX)
<xxx>_STATUS_MLT_TXQUEUE		(E4)	... any TX-Queue
<xxx>_STATUS_MLT_RXFIFO0		(F), (G)	RX FIFO 0
<xxx>_STATUS_MLT_RXFIFO1			RX FIFO 1
<xxx>_STATUS_MOW_COMFIFO	Message overwrite indication of dedicated reception target. Status value is set on overwrite event on...	(E4)	... any multi-purpose FIFO (RX)
<xxx>_STATUS_MOW_TXQUEUE			... any TX-Queue operated in gateway mode.
<xxx>_STATUS_BUSLOAD	Activates bus load measurement and returns bus load value of previous measurement.		The first readout has to be discarded, because the measurement is started by the same call of this function.
<xxx>_STATUS_TIMESTAMP	Current value of timestamp counter		*GTSC* register value of selected CAN controller unit.

a. By specifying <xxx>_GLOBAL as *ChannelNumber_u08*, the mode is returned for the global function of the selected CAN controller unit (*UnitNumber_u08*).
b. Apply one of these filter to status value: <xxx>_ERROR_WARNING, <xxx>_ERROR_PASSIVE, <xxx>_ERROR_BUSOFF.
c. Error detail information is one of: <xxx>_ERRORDETAIL_NONE (no error), <xxx>_ERRORDETAIL_STUFF (stuff error), <xxx>_ERRORDETAIL_FORM (form error), <xxx>_ERRORDETAIL_ACK (acknowledge error), <xxx>_ERRORDETAIL_BIT1 (bit 1 level error), <xxx>_ERRORDETAIL_BIT0 (bit 0 level error), <xxx>_ERRORDETAIL_CRC (CRC error).
d. In implementations (A) and (B), the label is not available and thus the box number is returned. In implementations (C), (D), (Ex) and (E4), the box number is returned, if the message transmit label was set to <xxx>_TID_NOTUSED; otherwise, the label is returned. In implementations (F) and (G), the message transmit label is returned.
e. In implementation (E4), the combination of RX-FIFO interrupt and RX-FIFO full interrupt is returned; the full interrupt flag is contained in the value right-shifted by <xxx>_RXFIFO_STATUS_FULL_P.

f. If *ChannelNumber_u08* is set to <xxx>_GLOBAL, the global interrupt flag status is returned within the status value; otherwise the channel interrupt flag status is returned.
Global interrupt flags are combined by *OR*: <xxx>_INT_GERR (global error), <xxx>_INT_RXF*n* (RX FIFO) with *n* = {0 ... 1} on (C), {0 ... 7} on (D), (Ex).
Channel interrupt flags are combined by *OR*: <xxx>_INT_TX (transmission complete), <xxx>_INT_TXA (transmission aborted), <xxx>_INT_TXQ (Queue), <xxx>_INT_CERR (channel error), <xxx>_INT_TXHL (history list), <xxx>_INT_RXCF (multi-purpose FIFO RX), <xxx>_INT_TXCF (multi-purpose FIFO TX).

5.3.7.5 <xxx>_GetFIFOStatus( )

Implementations: (C), (D), (Ex), (E4).

(1) Implementations: (C)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
PathType_u08:	Select FIFO type: <xxx>_PATH_RXFIFO (to read status of a RX FIFO) or <xxx>_PATH_COMFIFO (to read status of a multi-purpose FIFO)
FIFONumber_u08:	0 or 1 for <xxx>_PATH_RXFIFO or 0 for <xxx>_PATH_COMFIFO
FIFOEmpty_pu08:	Set, if the selected FIFO is empty
FIFOFull_pu08:	By reference; set, if the selected FIFO is full
FIFOMessageLost_pu08:	By reference; set, if the selected FIFO has lost a message
FIFOInterrupt_pu08:	By reference; set, if the selected FIFO has generated an interrupt
FIFOMessageCount_pu16:	By reference; Number of messages within the selected FIFO

(1-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(1-3) Functional Description

The function reports the current status of the addressed FIFO unit.

(2) Implementations: (D), (Ex), (E4)

(2-1) Parameters

UnitNumber_u08:	Selected CAN Controller
ChannelNumber_u08:	Selected CAN Controller channel
FIFONumber_u08:	Selected FIFO to be addressed
StatusType_u08:	Selected FIFO status to be checked
StatusValue_pu08:	Status return value returned by reference

(2-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(2-3) Functional Description

The function reports the current status of the addressed FIFO unit.
If the *ChannelNumber_u08* is set to <xxx>_GLOBAL, then the RX FIFO units are addressed, and the value range of *FIFONumber_u08* is within {0 ... 7}. Otherwise, for each selected channel, one of three multi-purpose FIFO units can be addressed by setting *FIFONumber_u08* in the range of {0 ... 2}.
Depending on *StatusType_u08*, the function returns the following status information by reference in *StatusValue_pu08*:

<xxx>_FIFO_STATUS_FULL:	Set, if the selected FIFO is full
<xxx>_FIFO_STATUS_EMPTY:	Set, if the selected FIFO is empty
<xxx>_FIFO_STATUS_LOST:	Set, if the selected FIFO has lost a message
<xxx>_FIFO_STATUS_LEVEL:	Set, if the selected FIFO's fill level has been reached (E4 only)
<xxx>_FIFO_STATUS_MCNT:	Returns selected FIFO's actual fill level (E4 only)
<xxx>_FIFO_STATUS_OVR:	Set, if in the selected FIFO a message was overwritten (E4 only, only for multi-purpose FIFO units)
<xxx>_FIFO_STATUS_OFRX:	Set, if the selected FIFO has received at least one message (E4 only, only for multi-purpose FIFO units in gateway mode)
<xxx>_FIFO_STATUS_OFTX:	Set, if the selected FIFO has transmitted at least one message (E4 only, only for multi-purpose FIFO units in gateway mode)

5.3.7.6 <xxx>_GetError( )

Implementations: (A), (B), (C), (D), (Ex), (E4).

(1) Implementations: (A), (B)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
MachineNumber_u08:	Selected CAN Controller channel
InterruptErrorFlag_pu08:	Return by reference: Driver error interrupt flag
LastMachineErrorFlag_pu08:	Return by reference: Last error of channel

(1-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(1-3) Functional Description

From the selected CAN controller unit and channel, the function returns by reference interrupt status and communication error statuses. The return value by *InterruptErrorFlag_pu08* contains the last value of the *INTS* register since the last error interrupt. The return value *LastMachineErrorFlag_pu08* reads out the *LEC* (last error code) of the channel, independently from any interrupt.

Values for *LEC* can be either:
<xxx>_LEC_OK (no error),
<xxx>_LEC_STUFF (stuff error),
<xxx>_LEC_FORM (form error),
<xxx>_LEC_ACK (acknowledge error),
<xxx>_LEC_BITREC (recessive bit error),
<xxx>_LEC_BITDOM (dominant bit error),
<xxx>_LEC_CRC (CRC error).

The error state in hardware is cleared by this function call.

(2) Implementations: (C), (D), (Ex), (E4)

(2-1) Parameters

UnitNumber_u08:	Selected CAN Controller
ChannelNumber_u08:	Selected CAN Controller channel
InterruptErrorFlag_pu16:	Return by reference: Driver error interrupt flag
LastErrorFlag_pu16:	Return by reference: Last error of channel

(2-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(2-3) Functional Description

From the selected CAN controller unit and channel, the function returns by reference communication error status information. While *InterruptErrorFlag_pu16* returns the last communication error status after the lastly occurred interrupt, *LastErrorFlag_pu16* returns the same directly read from hardware, independently from any interrupt.

If *ChannelNumber_u08* is specified by <xxx>_GLOBAL, the following error codes are returned and

bitwise combined:
<xxx>_GERROR_DLCCHECK (DLC too small error),
<xxx>_GERROR_MSGLOST (message lost error - implementations (C), (D), (Ex) only),
<xxx>_GERROR_MLT_TXFIFO (message lost error - implementation (E4) only),
<xxx>_GERROR_MLT_TXQUEUE (message lost in TX-Queue error - implementation (E4) only),
<xxx>_GERROR_THLLOST (transmit history list overflow error),
<xxx>_GERROR_RAMPARITY (RAM parity error - implementation (C) only),
<xxx>_GERROR_PLLFAIL (clocking failure error - implementations (C), (D), (Ex) only),
<xxx>_GERROR_OTBFAIL (transfer layer buffer failure error)
<xxx>_GERROR_FDMSGOVF (CAN-FD payload overflow error - implementation (E4) only),
<xxx>_GERROR_MOW_TXFIFO (multi-purpose FIFO message overwrite event - (E4) only),
<xxx>_GERROR_MOW_TXQUEUE (TX-Queue message overwrite event - (E4) only).

Otherwise, channel specific error status is returned and bitwise combined:
<xxx>_ERROR_BUSERR (CAN bus error),
<xxx>_ERROR_WARNING (error warning level reached),
<xxx>_ERROR_PASSIVE (error passive level reached),
<xxx>_ERROR_BUSOFF (bus off state reached),
<xxx>_ERROR_RECOVERY (channel is in bus off recovery state),
<xxx>_ERROR_OVERLOAD (an overload flag was detected on the CAN bus),
<xxx>_ERROR_BUSLOCK (the CAN bus was locked dominant excessively long),
<xxx>_ERROR_ARBLOST (arbitration lost),
<xxx>_ERROR_STUFFING (stuff error detected),
<xxx>_ERROR_FORM (form error detected),
<xxx>_ERROR_ACK (missing acknowledge error),
<xxx>_ERROR_CRC (CRC error detected),
<xxx>_ERROR_BITLEV1 (recessive bit error detected),
<xxx>_ERROR_BITLEV0 (dominant bit error detected),
<xxx>_ERROR_ACKDELIM (acknowledge delimiter error detected).

The error state in hardware is cleared by this function call.

5.3.7.7 <xxx>_GetTimeStampCounter( )

Implementations: (D), (Ex), (E4).

(1) Implementations: (D), (Ex), (E4)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
TimeStampValue_pu32:	Return value by reference: Timestamp counter value

(1-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(1-3) Functional Description

From the selected CAN controller unit and channel, the function returns by reference the current count value of the timestamp counter.

5.3.8 Transmission and Reception

5.3.8.1 <xxx>_SetSendMessage( )

Implementations: (A), (B).

(1) Implementations: (A), (B)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
MachineNumber_u08:	Selected CAN Controller channel
BufferNumber_u08:	Selected message buffer number to be used for transmission
ExtendedFrame_u08:	Extended frame format specification
RemoteFrame_u08:	Remote frame selection
InterruptEnable_u08:	Transmit interrupt activation for this buffer
DataLength_u08:	Data length code (*DLC*) of the message to be sent
Identifier_u32:	Identifier (*ID*) of the message to be sent
DataField_pu08:	Pointer to data buffer of the message to be sent

(1-2) Return Values

<xxx>_ERROR on parameter failures or hardware timeout, otherwise *<xxx>_OK*.

(1-3) Functional Description

Using the selected CAN controller unit, channel and message buffer, the function prepares a message to be sent out. Transmission of the message is not performed by this function.
The data buffer pointed to by *DataField_pu08* must provide enough data bytes, as they are specified by the data length code of *DataLength_u08*. The maximum data length is <xxx>_DLC_MAX.
If the frame format of *ExtendedFrame_u08* is set to <xxx>_ID_STD, the value range of *Identifier_u32* is {0 ... 0x7FF}; if it is set to <xxx>_ID_EXT, then the value range of *Identifier_u32* is {0 ... 0x3FFFFFFF}. Any shifting of the identifier in case of standard frames is not required, this is performed by the function.
If remote frames shall be sent instead of data frames, then *RemoteFrame_u08* shall be set to 1, otherwise to 0. In case of remote frames, the values for *DataLength_u08* and *DataField_pu08* are still required and need to be provided properly with safe dummy values at least.
If a transmit interrupt shall be generated after successful transmission of the message, then the flag *InterruptEnable_u08* has to be set 1, otherwise 0. Note that interrupt generation also depends on other settings, see functions *<xxx>_CreateInterrupt( )* and *<xxx>_SetInterrupt( )*.

5.3.8.2 <xxx>_SetReceiveMessage( )

Implementations: (A), (B).

(1) Implementations: (A), (B)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
MachineNumber_u08:	Selected CAN Controller channel
BufferNumber_u08:	Selected message buffer number to be used for transmission
ExtendedFrame_u08:	Extended frame format specification
RemoteFrame_u08:	Remote frame selection
InterruptEnable_u08:	Transmit interrupt activation for this buffer
OverWriteEnable_u08:	Overwrite enable setting for this buffer
BufferMaskFlag_u08:	Global reception mask selection for this buffer
Identifier_u32:	Identifier (*ID*) of the message to be sent

(1-2) Return Values

<xxx>_ERROR on parameter failures or hardware timeout, otherwise *<xxx>_OK*.

(1-3) Functional Description

Using the selected CAN controller unit, channel and message buffer, the function prepares a message buffer for reception.
If the frame format of *ExtendedFrame_u08* is set to <xxx>_ID_STD, the value range of *Identifier_u32* is {0 ... 0x7FF}; if it is set to <xxx>_ID_EXT, then the value range of *Identifier_u32* is {0 ... 0x3FFFFFFF}. Any shifting of the identifier in case of standard frames is not required, this is performed by the function.
If remote frames shall be received instead of data frames, then *RemoteFrame_u08* shall be set to 1, otherwise to 0.
If a reception interrupt shall be generated after a successful reception within this message buffer, then the flag *InterruptEnable_u08* has to be set 1, otherwise 0. Note that interrupt generation also depends on other settings, see functions *<xxx>_CreateInterrupt( )* and *<xxx>_SetInterrupt( )*.
If the *OverWriteEnable_u08* is not set, a reception within a buffer is locking the buffer, until read by using *<xxx>_ReceiveMessage()*; thus, an interrupt would be generated only once and new receptions would not take place until the buffer is read.
If the *OverWriteEnable_u08* is set, any new message which matches the reception criteria (matching mask combined with ID) may overwrite the buffer at any time. Thus, reading the buffer by using *<xxx>_ReceiveMessage()* would always yield the last message which matched and was received.
The *BufferMaskFlag_u08* is specifying the assigned global reception mask for this buffer. The masks can be set by using *<xxx>_SetMachineMask()*. The mask number shall be addressed by using the constants <xxx>_MASK_*n*, where the range of *n* depends on the implementation: For [A], *n* = {0 ... 3}, for [B], *n* = {0 ... 7}.

5.3.8.3 <xxx>_SendMessage( )

Implementations: (A), (B), (C), (D), (Ex), (E4), (F), (G).

(1) Implementations: (A), (B)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
MachineNumber_u08:	Selected CAN Controller channel
BufferNumber_u08:	Selected message buffer number to be used for transmission
AbortFlag_u08:	Aborts transmission if pending in selected buffer

(1-2) Return Values

<xxx>_ERROR on parameter failures, when called in *Initialization Mode*, or when trying to abort a transmission of a buffer, where no transmission is pending or ongoing. Otherwise *<xxx>_OK*.

(1-3) Functional Description

Using the selected CAN controller unit, channel and message buffer, the function triggers a message to be sent out. If a transmission is pending for a message buffer, the transmission can be aborted by setting *AbortFlag_u08* to 1. As the function will return <xxx>_ERROR, if an abortion is tried on a free buffer, the function can be used to find a free buffer by this, followed by a second call for a transmission on the same buffer, if it was found to be free.
If *MachineNumber_u08* is set to <xxx>_GLOBAL, then the *ABT Mode* trigger is set instead of a trigger of a dedicated message buffer. In this case, the *BufferNumber_u08* is not relevant.

For implementations [A] and [B], the function is covering the transmit abortion functionality, which is not available with the API *<xxx>_TxAbort( )*.

(2) Implementations (C), (D), (Ex), (E4)

(2-1) Parameters

UnitNumber_u08:	Selected CAN Controller
ChannelNumber_u08:	Selected CAN Controller channel
Status_pu08:	Status return value by reference
Message:	Pointer to structure of message to be loaded and sent Structure name is <*xxx*>_*message*, where <xxx> is in lower case.

(2-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(2-3) Functional Description

Using the selected CAN controller unit and channel, the function loads a message and triggers its transmission. Depending on parameters within the <*xxx*>_*message* structure, the function uses certain hardware resources or searches for free resources to perform a transmission.
The function returns a status by reference, contained in the location of *Status_pu08*. The following status return values are provided:

<xxx>_FAULT_NONE	Transmission request accepted
<xxx>_FAULT_PARAMETER	Unit, channel or message path details are out of range
<xxx>_FAULT_BUSY	Desired transmission path is currently busy
<xxx>_FAULT_ECCERR	Transmission not started because of ECC errors (implemented in (E4) only)

The <*xxx*>_*message* structure contains the following elements:

Table 5.19 Transmit Message Structure Elements (C), (D), (E2)

Element	Type	Value Range	Description
hdr [Message header]	<xxx>_t_mask.id <xxx>_t_mask.thlen <xxx>_t_mask.rtr <xxx>_t_mask.ide	{0 ... 0x1FFFFFFF} {0 , 1} {0 , 1} {0 , 1}	Identifier of message to be sent. (a) Entry in transmit history list after successful transmission, if set. Transmits a remote frame, if set. Transmits an extended ID frame, if set.
flag [Additiional flags and header info]	<xxx>_r_ptr.ts <xxx>_r_ptr.ptr <xxx>_r_ptr.dlc	- {0 ... 0xFFF} {0 ... 8} {0 ... 15}	Not used for transmission. Message label for transmit history list. Message length code. [C], [D] Message length code. [E] (b)
fdsts [CAN-FD information] (E2) only	<xxx>_r_fdsts.esi <xxx>_r_fdsts.brs <xxx>_r_fdsts.fdf	{0 , 1} {0 , 1} {0 , 1}	Error state indicator. (c) Use the data bit rate switching, if set. Use the CAN-FD frame format, if set. If not set, classical CAN frame format is used.
data [Message data]	u08[<xxx>_DLC_MAX] u08[<xxx>_DLC_FDMAX]	{0 ... 0xFF}	Data array of bytes to be sent. [C], [D] Data array of bytes to be sent. [E]
path [Transmission path]	u08	Dedicated values (d)	Specification from which resource to transmit the message.
pathdetail [Transmission path detail]	u08	Depends on path (e)	Specific path resource to use for transmission. Use <xxx>_PATHDETAIL_ANY wildcard to choose the first free resource starting from zero.

a. For standard ID messages, the value range is restricted to {0 ... 0x7FF}.
b. The length code has to be specified according to ISO 11898-1.
c. If set, message is sent with ESI=1. If not set, message is sent according to the channel's error passive status (ISO).
d. <xxx>_PATH_MSGBOX: Transmit from a free message box.
<xxx>_PATH_COMFIFO: Transmit from a multi-purpose FIFO in transmit mode.
<xxx>_PATH_TXQUEUE: Transmit from a transmit queue.
<xxx>_PATH_ANY: Check all transmit resources in the sequence as given above for a free resource.
e. If path is <xxx>_PATH_MSGBOX: Value range is {0 ... <xxx>_MAXTXBUFFERS-1} or wildcard in non-merged TX mode.
Value range is {0 ... <xxx>_MAXTXMBUFFERS-1} or wildcard in merged TX mode.
See 5.3.5.3, (E), for TX merge mode configuration in [E] implementations.
If path is <xxx>_PATH_COMFIFO: Value range is {0 ... MAXCOMFIFOS-1} or wildcard.
If path is <xxx>_PATH_TXQUEUE: Value range is {0 ... MAXTXQUEUES-1} or wildcard.
If path is <xxx>_PATH_ANY: Wildcard value of <xxx>_PATHDETAIL_ANY is allowed only.

Table 5.20 Transmit Message Structure Elements (E3), (E4)

Element	Type	Value Range	Description
hdr [Message header]	<xxx>_t_mask.id <xxx>_t_mask.thlen <xxx>_t_mask.rtr <xxx>_t_mask.ide	{0 ... 0x1FFFFFFF} {0 , 1} {0 , 1} {0 , 1}	Identifier of message to be sent. (a) Entry in transmit history list after successful transmission, if set. Transmits a remote frame, if set. Transmits an extended ID frame, if set.
flag [Additiional flags and header info]	<xxx>_r_ptr.ts <xxx>_r_ptr.dlc	- {0 ... 8} {0 ... 15}	Not used for transmission. Message length code. (C), (D) Message length code. (Ex) (b)
fdsts [CAN-FD information]	<xxx>_r_fdsts.esi <xxx>_r_fdsts.brs <xxx>_r_fdsts.fdf <xxx>_r_fdsts.ptr <xxx>_r_fdsts.ifl	{0 , 1} {0 , 1} {0 , 1} {0 ... 0xFFFF} {0 ... 3}	Error state indicator. (c) Use the data bit rate switching, if set. Use the CAN-FD frame format, if set. If not set, classical CAN frame format is used. Message label for transmit history list. Additional filtering label for AFL when transmitting in gateway mode ((E4) only).
data [Message data]	u08[<xxx>_DLC_FDMAX]	{0 ... 0xFF}	Data array of bytes to be sent.

Table 5.20 Transmit Message Structure Elements (E3), (E4)

Element	Type	Value Range	Description
path [Transmission path]	u08	Dedicated values (d)	Specification from which resource to transmit the message.
pathdetail [Transmission path detail]	u08	Depends on path (e)	Specific path resource to use for transmission. Use <xxx>_PATHDETAIL_ANY wildcard to choose the first free resource starting from zero.

a. For standard ID messages, the value range is restricted to {0 ... 0x7FF}.
b. The length code has to be specified according to ISO 11898-1.
c. If set, message is sent with ESI=1. If not set, message is sent according to the channel's error passive status (ISO).
d. <xxx>_PATH_MSGBOX: Transmit from a free message box.
<xxx>_PATH_COMFIFO: Transmit from a multi-purpose FIFO in transmit mode.
<xxx>_PATH_TXQUEUE: Transmit from a transmit queue.
<xxx>_PATH_ANY: Check all transmit resources in the sequence as given above for a free resource.
e. If path is <xxx>_PATH_MSGBOX: Value range is {0 ... <xxx>_MAXTXBUFFERS-1} or wildcard.
If path is <xxx>_PATH_COMFIFO:Value range is {0 ... MAXCOMFIFOS-1} or wildcard.
If path is <xxx>_PATH_TXQUEUE: Value range is {0 ... MAXTXQUEUES-1} or wildcard.
If path is <xxx>_PATH_ANY: Wildcard value of <xxx>_PATHDETAIL_ANY is allowed only.

(3) Implementations (F), (G)

(3-1) Parameters

UnitNumber_u08:	Selected CAN Controller
Status_pu08:	Status return value by reference
Message:	Pointer to structure of message to be loaded and sent Structure name is <*xxx*>_*tx*, where <xxx> is in lower case.

(3-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(3-3) Functional Description

Using the selected CAN controller unit, the function loads a message and triggers its transmission. Depending on parameters within the <*xxx*>_*tx* structure, the function uses certain hardware resources or searches for free resources to perform a transmission.
The function returns a status by reference, contained in the location of *Status_pu08*. The following status return values are provided:

<xxx>_FAULT_NONE	Transmission request accepted
<xxx>_FAULT_PARAMETER	Unit or message path details are out of range
<xxx>_FAULT_BUSY	Desired transmission path is currently busy

The <*xxx*>_*tx* structure contains the following elements:

Table 5.21 Transmit Message Structure Elements (F), (G)

Element	Type	Value Range	Description
t0 [Message header 1st part]	<xxx>_tx_t0.id <xxx>_tx_t0.rtr <xxx>_tx_t0.xtd	{0 ... 0x1FFFFFFF} {0 , 1} {0 , 1}	Identifier of message to be sent. (a) Transmits a remote frame, if set. Transmits an extended ID frame, if set.
t1 [Message header 2nd part]	<xxx>_tx_t1.dlc <xxx>_tx_t1.efc <xxx>_tx_t1.mm	{0 ... 8} {0 ... 15} {0 ... 1} {0 ... 0xFF}	Message length code for classical CAN. Message length code for CAN-FD frames. (b) Entry in transmit event list after successful transmission, if set. Message label for transmit event list.
data [Message data]	union of access sizes: lw [<xxx>_DLC_FDMAX/4] sw [<xxx>_DLC_FDMAX/2] b [<xxx>_DLC_FDMAX]	 {0 ... 0xFFFFFFFF} {0 ... 0xFFFF} {0 ... 0xFF}	 Data array of long words to be sent. Data array of short words to be sent. Data array of bytes to be sent.
path [Transmission path]	u08	Dedicated values (c)	Specification from which resource to transmit the message.
pathdetail [Transmission path detail]	u08	Depends on path (d)	Specific path resource to use for transmission. Use <xxx>_PATHDETAIL_ANY wildcard to choose the first free resource starting from zero.

a. For standard ID messages, the value range is restricted to {0 ... 0x7FF}.
b. The length code has to be specified according to ISO 11898-1.
c. <xxx>_PATH_MSGBOX: Transmit from a free message box.
<xxx>_PATH_FIFOQUEUE: Transmit from a transmit queue or FIFO (availability depends on configuration).
<xxx>_PATH_ANY: Check all transmit resources in the sequence as given above for a free resource.
d. If path is <xxx>_PATH_MSGBOX: Value range is {0 ... <xxx>_MAXTXBUFFERS-1} or wildcard.
The availability of a dedicated message box depends upon the configuration.
If path is <xxx>_PATH_FIFOQUEUE: Parameter is not used.
If path is <xxx>_PATH_ANY: Wildcard value of <xxx>_PATHDETAIL_ANY is allowed only.

5.3.8.4 <xxx>_ReceiveMessage( )

Implementations: (A), (B), (C), (D), (Ex), (E4), (F), (G).

(1) Implementations: (A), (B)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
MachineNumber_u08:	Selected CAN Controller channel
BufferNumber_u08:	Selected message buffer number to be read
TimeStampValue_pu16:	Timestamp of reception - currently not supported
NewDataFlag_pu08:	New reception data indication return by reference
OverwriteFlag_pu08:	Overwritten buffer indication return by reference
ReservedBit0_pu08:	R0 flag of received message - currently not supported
ReservedBit1_pu08:	R1 flag of received message - currently not supported
DataLength_pu08:	Received message data length return by reference
ExtendedFrame_pu08:	Received message frame type return by reference
Identifier_pu32:	Received message identifier return by reference
RemoteFrameReceived_pu08:	Received message type return by reference - only supported for [B]
DataField_pu08:	Received message data return by reference

(1-2) Return Values

<xxx>_ERROR on parameter failures, or when called in *Reset, Sleep or Stop Mode*. Otherwise *<xxx>_OK*.

(1-3) Functional Description

Using the selected CAN controller unit, channel and message buffer, the function reads a message from the dedicated message buffer.
The values *NewDataFlag_pu08, OverwriteFlag_pu08, DataLength_pu08, ExtendedFrame_pu08, Identifier_pu32, RemoteFrameReceived_pu08* ([B] only) and *DataField_pu08* are read from the message buffer hardware and returned by reference.
From processing point of view, the function first checks for a *NewDataFlag*, clears it in the hardware and then reads out the data. If, at the completion of the reading process, the *NewDataFlag* is again set, then the function repeats the reading process, because then the data has been overwritten meanwhile. As a result, the function always returns the newest data of the most recently received frames.
The values of the *Identifier_pu32* are plain values which do not need to be shifted in case of any frame format.
For *DataField_pu08*, the user must take care for appropriate memory space to write the data to.
At the returning of the function, the dedicated message buffer is again free to receive new messages.

(2) Implementations: (C), (D), (Ex), (E4)

(2-1) Parameters

UnitNumber_u08:	Selected CAN Controller
Status_pu08:	Reception status returned by reference
Message:	Pointer to structure of message received Structure name is <*xxx*>_*message*, where <xxx> is in lower case.

(2-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(2-3) Functional Description

Using the selected CAN controller unit, the function receives a message from any or a selectable resource. Depending on parameters within the <*xxx*>_*message* structure, the function uses certain hardware resources or searches for resources with new data to receive a new message.
The function returns a status by reference, contained in the location of *Status_pu08*. The following status return values are provided:

<xxx>_FAULT_NONE	New data is available and has been provided with *Message*.
<xxx>_FAULT_PARAMETER	Unit or message path details are out of range, or the controller is not in an operating mode.
<xxx>_FAULT_NODATA	No new data is available within the specified resource(s).

The <*xxx*>_*message* structure is used both for resource specification *and* for return of the messages. It contains the following elements:

Table 5.22 Receive Message Structure Elements (C), (D), (E2)

Element	Type	Value Range	Description
hdr [Message header]	<xxx>_t_mask.id <xxx>_t_mask.thlen <xxx>_t_mask.rtr <xxx>_t_mask.ide	{0 ... 0x1FFFFFFF} - {0 , 1} {0 , 1}	Identifier of message received. (a) Not used for reception. Remote frame received, if set. Extended ID frame received, if set.
flag [Additiional flags and header info]	<xxx>_r_ptr.ts <xxx>_r_ptr.ptr <xxx>_r_ptr.dlc	{0 ... 0xFFFF} {0 ... 0xFFF} {0 ... 8} {0 ... 15}	Reception timestamp counter value. Message label from matching AFL rule. Message length code. (C), (D) Message length code. (E2) (b)
fdsts [CAN-FD information] (E2) only	<xxx>_r_fdsts.esi <xxx>_r_fdsts.brs <xxx>_r_fdsts.fdf	{0 , 1} {0 , 1} {0 , 1}	Error state indicator within message. Data bit rate switching of message. CAN-FD frame format of message. If not set, the message is was received in classical CAN frame format.
data [Message data]	u08[<xxx>_DLC_MAX] u08[<xxx>_DLC_FDMAX]	{0 ... 0xFF}	Data array of bytes received. (C), (D) Data array of bytes received. (E2)
path [Reception path]	u08	Dedicated values (c)	Specification from which resource to receive a message.
pathdetail [Reception path detail]	u08	Depends on path (d)	Specific path resource to check for reception. Use <xxx>_PATHDETAIL_ANY wildcard to choose the first free resource starting from zero.

a. For standard ID messages, the value range is restricted to {0 ... 0x7FF}.
b. The length code is specified according to ISO 11898-1.
c. <xxx>_PATH_MSGBOX: Check reception of standard message box(es).
<xxx>_PATH_RXFIFO: Check reception of a common receive FIFO.
<xxx>_PATH_COMFIFO: Check reception of a multi-purpose FIFO in receive mode.
<xxx>_PATH_ANY: Check all reception resources in the sequence as given above for a reception.
d. If path is <xxx>_PATH_MSGBOX: Value range is {0 ... <xxx>_MAXRXBUFFERS-1} or wildcard.
If path is <xxx>_PATH_RXFIFO: Value range is {0 ... MAXRXFIFOS-1} or wildcard.
If path is <xxx>_PATH_COMFIFO: Value range is {0 ... MAXCOMFIFOS-1} or wildcard.
If path is <xxx>_PATH_ANY: Wildcard value of <xxx>_PATHDETAIL_ANY is allowed only.

Table 5.23 Receive Message Structure Elements (E3), (E4)

Element	Type	Value Range	Description
hdr [Message header]	<xxx>_t_mask.id <xxx>_t_mask.thlen <xxx>_t_mask.rtr <xxx>_t_mask.ide	{0 ... 0x1FFFFFFF} - {0 , 1} {0 , 1}	Identifier of message to be sent. (a) Not used for reception. Remote frame received, if set. Extended ID frame received, if set.
flag [Additiional flags and header info]	<xxx>_r_ptr.ts <xxx>_r_ptr.dlc	{0 ... 0xFFFF} {0 ... 15}	Reception timestamp counter value. Message length code. (b)
fdsts [CAN-FD information]	<xxx>_r_fdsts.esi <xxx>_r_fdsts.brs <xxx>_r_fdsts.fdf <xxx>_r_fdsts.ptr <xxx>_r_fdsts.ifl	{0 , 1} {0 , 1} {0 , 1} {0 ... 0xFFFF} {0 ,,, 3}	Error state indicator within message. Data bit rate switching of message. CAN-FD frame format of message. If not set, the message is was received in classical CAN frame format. Message label from matching AFL rule. Additional filtering label from matching AFL rule ((E4) only).
data [Message data]	u08[<xxx>_DLC_FDMAX]	{0 ... 0xFF}	Data array of bytes received.
path [Reception path]	u08	Dedicated values (c)	Specification from which resource to receive a message.

Table 5.23 Receive Message Structure Elements (E3), (E4)

Element	Type	Value Range	Description
pathdetail [Reception path detail]	u08	Depends on path (d)	Specific path resource to use for reception. Use <xxx>_PATHDETAIL_ANY wildcard to choose the first free resource starting from zero.

a. For standard ID messages, the value range is restricted to {0 ... 0x7FF}.
b. The length code has to be specified according to ISO 11898-1.
c. <xxx>_PATH_MSGBOX: Check reception of standard message box(es).
<xxx>_PATH_RXFIFO: Check reception of a common receive FIFO.
<xxx>_PATH_COMFIFO: Check reception of a multi-purpose FIFO in receive mode.
<xxx>_PATH_ANY: Check all reception resources in the sequence as given above for a reception.
d. If path is <xxx>_PATH_MSGBOX: Value range is {0 ... <xxx>_MAXRXBUFFERS-1} or wildcard.
If path is <xxx>_PATH_RXFIFO: Value range is {0 ... MAXRXFIFOS-1} or wildcard.
If path is <xxx>_PATH_COMFIFO: Value range is {0 ... MAXCOMFIFOS-1} or wildcard.
If path is <xxx>_PATH_ANY: Wildcard value of <xxx>_PATHDETAIL_ANY is allowed only.

(3) Implementations (F), (G)

(3-1) Parameters

UnitNumber_u08:	Selected CAN Controller
Status_pu08:	Status return value by reference
Message:	Pointer to structure of message received Structure name is <*xxx*>_*rx*, where <xxx> is in lower case.

(3-2) Return Values

<*xxx*>_*ERROR* on parameter failures or when called in *initialization* mode, otherwise <*xxx*>_*OK*.

(3-3) Functional Description

Using the selected CAN controller unit, the function receives a message from any or a selectable resource. Depending on parameters within the <*xxx*>_*rx* structure, the function uses certain hardware resources or searches for resources with new data to receive a new message.
The function returns a status by reference, contained in the location of *Status_pu08*. The following status return values are provided:

<xxx>_FAULT_NONE	New data is available and has been provided with *Message*.
<xxx>_FAULT_PARAMETER	Unit or message path details are out of range.
<xxx>_FAULT_NODATA	No new data is available within the specified resource(s).

The <*xxx*>_*rx* structure contains the following elements:

Table 5.24 Receive Message Structure Elements (F), (G)

Element	Type	Value Range	Description
r0 [Message header 1st part]	<xxx>_rx_r0.id <xxx>_rx_r0.rtr <xxx>_rx_r0.xtd <xxx>_rx_r0.esi	{0 ... 0x1FFFFFFF} {0 , 1} {0 , 1} {0 , 1}	Identifier of message to be sent. (a) Remote frame received, if set. Extended ID frame received, if set. Error state indicator within message.
r1 [Message header 2nd part]	<xxx>_rx_r1.rxts <xxx>_rx_r1.dlc <xxx>_rx_r1.brs <xxx>_rx_r1.edl <xxx>_rx_r1.fidx <xxx>_rx_r1.anmf	{0 ... 0xFFFF} {0 ... 8} {0 ... 15} {0 , 1} {0 , 1} {0 , 0x7F} {0 , 1}	Reception timestamp counter value. Message length code for classical CAN. Message length code for CAN-FD frames. (b) Data bit rate switching of message. CAN-FD frame format of message. If not set, the message is was received in classical CAN frame format. Order number of reception rule which matched If set, no reception rule did match.
data [Message data]	union of access sizes: lw [<xxx>_DLC_FDMAX/4] sw [<xxx>_DLC_FDMAX/2] b [<xxx>_DLC_FDMAX]	 {0 ... 0xFFFFFFFF} {0 ... 0xFFFF} {0 ... 0xFF}	 Data array of long words received. Data array of short words received. Data array of bytes to be received.
path [Transmission path]	u08	Dedicated values (c)	Specification from which resource to transmit the message.
pathdetail [Transmission path detail]	u08	Depends on path (d)	Specific path resource to use for transmission. Use <xxx>_PATHDETAIL_ANY wildcard to choose the first free resource starting from zero.

a. For standard ID messages, the value range is restricted to {0 ... 0x7FF}.
b. The length code has to be specified according to ISO 11898-1.
c. <xxx>_PATH_MSGBOX: Check reception of standard message box(es).
<xxx>_PATH_FIFOQUEUE: Check reception of a FIFO (availability depends on configuration).
<xxx>_PATH_ANY: Check all reception resources in the sequence as given above for a reception.

d. If path is <xxx>_PATH_MSGBOX: Value range is {0 ... <xxx>_MAXRXBUFFERS-1} or wildcard.
The availability of a dedicated message box depends upon the configuration.
If path is <xxx>_PATH_FIFOQUEUE: Value range is {0 ... 1} or wildcard.
The availability of a dedicated FIFO depends upon the configuration.
If path is <xxx>_PATH_ANY: Wildcard value of <xxx>_PATHDETAIL_ANY is allowed only.

5.3.8.5 <xxx>_CheckReceiveMessage( )

Implementations: (A), (B).

(1) Implementations: (A), (B)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
MachineNumber_u08:	Selected CAN Controller channel
BufferNumber_pu08:	Preselected message buffer for *DNFlags_pu32* flag group selection Number of message buffer of oldest reception (return by reference)
StatusFlag_pu08:	Status return value by reference
DNFlags_pu32:	Map of *New Data* flags around the preselected buffer ([B] only)

(1-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(1-3) Functional Description

Using the selected CAN controller unit and channel, the function checks for any messages which have been received.
The function is using the *Receive History List* of the CAN controller hardware to check for any received messages and performs the history list handling appropriately.
Any message waiting for reception by software is indicated (oldest first) by its containing buffer (*BufferNumber_pu08*), so that in the following the function *<xxx>_ReceiveMessage( )* can be called to retrieve it.
Within implementations [B], the function additionally returns the *New Data* flags of a set of 32 buffers within *DNFlags_pu32*. As there may be more buffers than 32, the return value can be focused on a subset of buffers in the range {0 ... 31} or {32 ... 63} and so on. These ranges are preselected by pre-loading the value *BufferNumber_pu08* with a buffer number belonging to the range desired.

Depending on the *Receive History List* check, the function returns the following status values by reference within *StatusFlag_pu08*:

<xxx>_STATUS_ROVF:	The receive history list has overflown and now is consequently reset.
<xxx>_STATUS_RHPM:	No message has been received - the history list is empty.
<xxx>_STATUS_RGPT:	A message waits for processing in buffer *BufferNumber_pu08*.

If the function returns with *<xxx>_ERROR*, then the value of *StatusFlag_pu08* is invalid.

5.3.8.6 <xxx>_CheckSendMessage( )

Implementations: (A), (B).

(1) Implementations: (A), (B)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
MachineNumber_u08:	Selected CAN Controller channel
BufferNumber_pu08:	Number of message buffer of oldest transmission (return by reference)
StatusFlag_pu08:	Status return value by reference

(1-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(1-3) Functional Description

Using the selected CAN controller unit and channel, the function checks for any messages which have been transmitted successfully.
The function is using the *Transmit History List* of the CAN controller hardware to check for any transmitted messages and performs the history list handling appropriately.
Any message waiting for transmission recognition by software is indicated (oldest first) by its containing buffer (*BufferNumber_pu08*).

Depending on the *Transmit History List* check, the function returns the following status values by reference within *StatusFlag_pu08*:

<xxx>_STATUS_TOVF:	The transmit history list has overflown and now is consequently reset.
<xxx>_STATUS_THPM:	No message has been transmitted - the history list is empty.
<xxx>_STATUS_TGPT:	A message has been sent from buffer *BufferNumber_pu08*.

If the function returns with *<xxx>_ERROR*, then the value of *StatusFlag_pu08* is invalid.

5.3.8.7 <xxx>_ClearReadyMessage( )

Implementations: (A), (B).

(1) Implementations: (A), (B)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
MachineNumber_u08:	Selected CAN Controller channel
LowerBufferNumberBoundary_u08:	Lower buffer number of a selected buffer range
UpperBufferNumberBoundary_u08:	Upper buffer number of a selected buffer range

(1-2) Return Values

<xxx>_ERROR on parameter failures or hardware timeout, otherwise *<xxx>_OK*.

(1-3) Functional Description

Using the selected CAN controller unit and channel, the function clears both the *RDY* and *TRQ* flags of all hardware message buffers within the range, which is given by and including {*LowerBufferNumberBoundary_u08 ... UpperBufferNumberBoundary_u08*}.

RDY and *TRQ* are hardware semaphores, which ***always must be cleared*** for all used message buffers for transmission, before stopping communication (transition to *INIT* mode) or before performing a bus-off recovery.

Thus, the function must be used in every application, if a re-initialization or a bus-off recovery is implemented. If the function is not used nevertheless, a hardware failure may result.

5.3.8.8 <xxx>_TxAbort( )

Implementations: (C), (D), (Ex), (E4), (F), (G).

(1) Implementations: (C), (D), (Ex), (E4)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
ChannelNumber_u08:	Selected CAN Controller channel
Message:	Pointer to structure of message to be aborted Structure name is <*xxx*>_*message*, where <xxx> is in lower case.

(1-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(1-3) Functional Description

Using the selected CAN controller unit and channel, the function aborts the transmission of a message from a dedicated resource. Confirmation of abortion is given by interrupt. The resource selection is done by specifying the fields *path* and *pathdetail*. For *path*, the following options are available:

<xxx>_PATH_MSGBOX:	If *pathdetail* is set to <xxx>_PATHDETAIL_ANY, all messages in all transmit buffers are aborted. This option can also be used to flush a whole transmit queue. Otherwise, the dedicated transmit buffer given in *pathdetail* is addressed for abortion.
<xxx>_PATH_COMFIFO: ((Ex) only) ((E4): not implemented)	The value of *pathdetail* must not be set to <xxx>_PATHDETAIL_ANY. The related transmit buffer for the selected multi-purpose FIFO is addressed and its transmission is aborted. The multi-purpose FIFO will then continue with its next message to be transmitted.
<xxx>_PATH_TXQUEUE:	This option is not yet available and may lead to malfunction if used.

(2) Implementations: (F), (G)

(2-1) Parameters

UnitNumber_u08:	Selected CAN Controller
Message:	Pointer to structure of message to be aborted Structure name is <*xxx*>_*tx*, where <xxx> is in lower case.

(2-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(2-3) Functional Description

Using the selected CAN controller unit, the function aborts the transmission of a message from a dedicated resource. Confirmation of abortion is given by interrupt. The resource selection is done by specifying the fields *path* and *pathdetail*. For *path*, the following options are available:

<xxx>_PATH_MSGBOX:	If *pathdetail* is set to <xxx>_PATHDETAIL_ANY, all messages in all transmit buffers are aborted. Otherwise, the dedicated transmit buffer given in *pathdetail* is addressed for abortion.
<xxx>_PATH_FIFOQUEUE:	The value of *pathdetail* is ignored. The front-most buffer of the transmit FIFO is addressed and its transmission is aborted.

5.3.8.9 <xxx>_CheckAbortStatus( )

Implementations: (B).

(1) Implementations: (B)

(1-1) Parameters

UnitNumber_u08:	Selected CAN Controller
MachineNumber_u08:	Selected CAN Controller channel
BufferNumber_u08:	Selected message buffer number to be checked
StatusFlag_pu08:	Transmit abortion status of the selected buffer returned by reference

(1-2) Return Values

<xxx>_ERROR on parameter failures or when in *reset* or *sleep* modes, if the abortion status is yet unclear, or if no transmit abortion was requested on the selected buffer. Otherwise *<xxx>_OK*.

(1-3) Functional Description

Using the selected CAN controller unit and channel, the function checks both the *TRQ* and *TCP* flags of a dedicated hardware message buffer.

If a transmission abortion was already initiated for the selected message buffer beforehand, by using the *<xxx>_SendMessage( )* function, the status of the abortion process is returned by the *StatusFlag_pu08*:

<xxx>_TXDONE:	The message transmission could not be aborted and was successfully sent out.
<xxx>_TXABORTED:	The message transmission was successfully aborted.

5.3.9 Diagnosis and Self Test

5.3.9.1 <xxx>_IntCANBusActivate( )

Implementations: (D), (Ex), (E4).

(1) Implementations: (D), (Ex), (E4)

(1-1) Parameters

UnitNumber_u08: Selected CAN Controller

(1-2) Return Values

<*xxx*>_*ERROR* on parameter failures or when in any other mode than *global halt*. Otherwise <*xxx*>_*OK*. *Global halt* mode can be activated by using the <*xxx*>_*Stop( )* function, with argument <xxx>_GLOBAL as channel number and mode <xxx>_OPMODE_HALT.

(1-3) Functional Description

Using the selected CAN controller unit, all of its channels are getting internally connected. At the same time, the channels will not be available for external communication.

5.3.9.2 <xxx>_RAMTest( )

Implementations: (C), (D), (Ex), (E4).

(1) Implementations: (C), (D), (Ex), (E4)

(1-1) Parameters

UnitNumber_u08: Selected CAN Controller

(1-2) Return Values

<*xxx*>_*ERROR* on parameter failures, on RAM check failure or when in any other mode than *global halt*. Otherwise <*xxx*>_*OK*. *Global halt* mode can be activated by using the <*xxx*>_*Stop( )* function, with argument <xxx>_GLOBAL as channel number and mode <xxx>_OPMODE_HALT.

(1-3) Functional Description

Using the selected CAN controller unit, its local RAM storage is checked page by page by clearing status and walking-one test patterns.
During the test, any communication is not possible.
After executing this function, all channels, message storage and global settings of the CAN controller need to be reinitialized (see configuration functions).

The amount of RAM pages and the amount of cells within the last page of the RAM depends on the product (size of CAN controller RAM). Therefore, the following dedicated *#define* constants must be set in the driver's mapping definition (see 5.4 for details and file information):

#define <xxx>_RAMTEST_PAGES
#define <xxx>_RAMTEST_LASTPGENTRIES

5.4 Mapping of the Lower CAN Driver

5.4.1 Device Level

On device level, the type and amount of CAN controllers are defined. Depending on the implemented CAN controller type, the following two entries in the file *map_device.h* are set:

```
#define <xxx>_MACROS    (n)    // Number of CAN controllers
#define <xxx>_TYPE      (m)    // Type of CAN controllers
```

Within the driver package, these lines are already filled in and do not require any change. For information, the available CAN controller types are listed in the common resource file *ree_macros.h*, which is also included in the driver packages.

5.4.2 CAN Controller IP Level

On IP level, device specific properties of the implemented CAN controller type(s) are defined in the files *map_*.h*. The file name depends on the CAN controller type:

<xxx> is ...		
	EE_AFCAN:	map_afcan.h
	EE_RSCAN:	map_rscan.h
	EE_RSCFD or RSCFD:	map_rscfd.h
	MCAN:	map_mcan.h
	MTTCAN:	map_mttcan.h

Within the *map_*.h* files, the following subsection will explain the various settings, and how to adapt them to individual needs. It is not recommended to change any other parameters of these files, which are not documented below.

5.4.2.1 Base Addresses

It is not recommended to change the base addresses. These are defining how and where the CAN controller and also some functional sub-parts are mapped within the device memory. The following entries can be found:

```
#define <xxx>_BASE
#define <xxx>_OFFSET_<...>
```

5.4.2.2 Device and Usage Adaptation

Several parameters exist, which may be altered by the user. Not all implementations are supporting every parameter.

(1) <xxx>_FREQFACTOR
This entry defines the factor between the given system clock frequency *OSCILLATOR_FREQUENCY* and the communication clock of the CAN controller. Typically, the factor is 1 for *AFCAN* and *M(TT)CAN* implementations and 0.5 for *RS-CAN(FD)* implementations. If a PLL is enabled, its frequency should be considered within the *OSCILLATOR_FREQUENCY* setting. The settings for *OSCILLATOR_FREQUENCY* are located in the configuration file *map_asmn_basic.h* (for the serial monitor program) or *map_tgmn.h* (for the graphics monitor program) respectively.

(2) <xxx>_MAXBAUDRATE
Defines the maximum selectable bit rate. This depends on the communication clock speed and (for some

implementation) on a minimum factor *<xxx>_CLKMINFACTOR*. The minimum factor may be required and is an implicit property of the CAN controller. Changes on this entry may cause under-clocking of the CAN controller and thus bit timings which are out of range.

(3) <xxx>_FREQFACTORPLLBP and <xxx>_MAXBAUDRATEPLLBP
These entries are defining the clock frequency factor and maximum bit rates in case of direct oscillator clock usage for communication. The direct oscillator clock is an optional clock which can be selected upon global configuration of the *RS-CAN* and *RS-CANFD* controllers. An adjustment of the clock factor may be required depending on the chosen oscillator frequency.

(4) <xxx>_SHUTDOWNTIMEOUT
The value indicates the amount of idling loops, while waiting on hardware reactions. This timeout supervision is mostly used when starting or shutting down, or when accessing common resources within a CAN controller, which are protected by semaphore mechanisms. Depending on the speed of the CPU, it may be required to adjust the value.

(5) <xxx>_VERBOSE
Activates debugging messages on the console or monitor serial interface, if set to one. This setting is not used by the driver, but most of all the application examples.

(6) <xxx>_PORT ...
Used to set the port I/O properties of the CAN controller and all of its channels. Redirection to dedicated pins and ports is done by setting these constants. See *5.3.4.1 <xxx>_PortEnable( )* for details.

(7) <xxx>_INT_BUNDLINGHOOK
If a CAN controller provides internal bundling of interrupt sources, so that less resources are required by the interrupt controller, then this variable is set to an additional support function, which is device specific and decodes the bundled interrupts. The variable can be used to insert additional interrupt execution.

(8) <xxx>_INT ...
Used to associate interrupt sources of a CAN controller with interrupt controller registers, which are used to enable or disable an interrupt. See *5.3.5.5 <xxx>_CreateInterrupt( )* for details.

(9) <xxx>_INTCLEAR
Constant to write into interrupt controller registers, if an interrupt shall be disabled. See the CPU core manual for proper values.

(10) <xxx>_INTENABLEDEFAULT
Constant to write into interrupt controller registers, if an interrupt shall be enabled. Not used by the lower driver by itself, but by application examples. See the CPU core manual for proper values.

(11) <xxx>_RAMTEST_PAGES and <xxx>_RAMTEST_LASTPGENTRIES
Amount of pages of the local RAM of the CAN controller. Used for RAM testing functions. See *5.3.9.2 <xxx>_RAMTest( )* for details.

5.4.2.3 Memory Vectors

In the last section of the *map*.h* file, the base addresses are applied to structures and arrays, in order to provide register access to the hardware for the low level driver.
These *static struct <xxx>_...* entries are specific for each CAN controller type and must not be altered by the user. Configuration and tailoring of the specific implementation to the device is done here, too (i.e., amount of channels, number of units).

5.5 Applications Based on the Lower CAN Driver

5.5.1 Serial Monitor Program

The serial monitor program is allowing the interactive execution of the application examples, as described in chapter *5.5.3 Communication Application Examples*, by commands from a TTY terminal. If supplied with the software package, the monitor program resides in the files *asmn*.c*.

Depending on implemented serial interfaces and the user's preference, the command console can be executed using certain resources. Within the file *map_asmn_basic.h*, the command console is directed to a resource by the setting of *ASMN_UARTINTERFACE*. The following UART types are compatible with the application examples (availability depends on the device):

V850 series:
UARTA_STANDARD, UARTC_STANDARD, UARTD_STANDARD, UARTE_STANDARD

78K0R, RL78 series:
UARTF_STANDARD

RH850 series:
RLIN3_STANDARD

All series with debug system only (using the debugger terminal console):
DEBUG_INTERNAL

5.5.1.1 Using the Debugger Console

If the debugger terminal console shall be used, besides the setting above, the following settings shall be made in the configuration file *map_asmn.h*:

```
#define ASMN_MENUCOM                    ( 0 )
#define ASMN_UARTTRANSFEROK             ( true )
#define ASMN_UARTERROR                  ( false )
#define ASMN_MENUCOM_EXE1               ( 0x0D )
#define ASMN_MENUCOM_EXE2               ( 0x0A )
#define ASMN_MENUCOM_BUFLEN             ( 4 )
#define ASMN_UARTMODECRLF               ( 1 )
#define ASMN_UARTMODECR                 ( 0 )

#define ASMN_UARTMODEFORCELF

#define ASMN_UARTSENDSTRING             ASMN_SendString
#define ASMN_UARTSENDBYTE               ASMN_SendByte
#define ASMN_UARTRECEIVEBYTE            ASMN_ReceiveByte
#define ASMN_UARTRECEIVEINT             ASMN_ReceiveInteger
#define ASMN_UARTRECEIVELONG            ASMN_ReceiveULong
#define ASMN_UARTRECEIVEFLOAT           ASMN_ReceiveFloat
```

This will redirect all UART communication functions to internal functions of the monitor program.

5.5.1.2 Using a Serial Interface

Depending on the serial interface specified in *map_asmn_basic.h*, the corresponding low level driver of the interface and its device specific mappings have to be added to the project.

A *map*.h* file also belongs to the low level driver of the serial interface. This file has to be configured similarly like the mapping file of the low level CAN driver. Here, the appropriate port and interrupt associations have to be set, so that it fits with the hardware environment. As an example, for a serial interface called *RLIN3*, the mapping file *map_rlin3.h* needs to be configured properly. See the mapping of the CAN driver in *5.4 Mapping of the Lower CAN Driver* section to have a generic overview.

In addition, the file *map_asmn.h* contains the serial interface parameters.

#define ASMN_MENUCOM	RLIN3_0	Serial Unit (here: *RLIN30*)
#define ASMN_MENUCOM_ILEVEL	0	Interrupt Level
#define ASMN_MENUCOM_BAUD	9600L	Bit rate
#define ASMN_MENUCOM_PARITY	RLIN3_PARITY_NONE	Parity setting
#define ASMN_MENUCOM_CHLEN	RLIN3_CHARLEN_8BITS	Bits per frame
#define ASMN_MENUCOM_STOP	RLIN3_ONESTOPBIT	Stop bits per frame
#define ASMN_MENUCOM_DIR	RLIN3_DIR_LSBFIRST	Bit ordering
#define ASMN_MENUCOM_EXE1	0x0D	First code for <return>
#define ASMN_MENUCOM_EXE2	0x0A	Alternative code
#define ASMN_MENUCOM_BUFLEN	4	Command buffer size

The monitor calls to initiate the user interface dialogue are mapped to the low level driver functions of the used serial interface. This is done with the set of *ASMN_UART...* definitions. When changing the console, the new driver functions can be entered here.

5.5.2 Graphics Monitor Program

For dedicated hardware of Renesas product series on certain application boards made by Renesas, where a graphics display with touchscreen can be installed, there is a graphics monitor program available for demonstration purposes.

The graphics monitor program is allowing the interactive execution of the application examples, as described in chapter *5.5.3 Communication Application Examples*, by commands from a touchscreen. If supplied with the software package, the monitor program resides in the files *tgmn*.c*.

Within the file *map_tgmn.h*, the graphics monitor program is configured regarding its underlying graphics display routines. It includes libraries from *zlib* and *libpng*, which are licensed under the premises of the *Open Source Initiative* and *Free Software Foundation*. Corresponding disclaimers and license text can be found in *5.5.2.1 Public Licenses of Graphics Routines*.

The graphics monitor program requires an additional hardware dependent support package, which is located in the files "*bsp_tgmn.**", where settings of physical connections and required additional resources (timer) are defined.

Using the structures in the files "*tgmn_<xxx>_tgmnif.h*", the application example functions are called by vectors and parameters gathered from the graphical user interface.

Using the timer activated in the hardware dependent support package, the touchscreen is checked for any input events of the GUI, and any pending GUI updates are executed.

5.5.2.1 Public Licenses of Graphics Routines

(1) zlib

Copyright (c) 1995~2017 Jean-Loup Gailly, Mark Adler

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

(1-1) The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
(1-2) Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
(1-3) This notice may not be removed or altered from any source distribution.

(2) libpng

Copyright (c) 1996~2017 Guy Eric Schalnat, Andreas Dilger, Glenn Randers-Pehrson et al.

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

(2-1) The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
(2-2) Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
(2-3) This notice may not be removed or altered from any source distribution.

5.5.3 Communication Application Examples

The application examples of the CAN driver reside in the files **_a.c* with API **_a.h*. The functions are called by the monitor program - so that the parameters can be entered via the terminal console.

Even though for some CAN controllers there may be more examples, the minimum equipment for a multi-channel CAN controller is a basic testing function, namely *<xxx>_BasicTest( )*.
If a CAN controller implementation only provides one channel, then the basic testing is split into 2 parts, *<xxx>_BasicTest_Rx( )* and *<xxx>_BasicTest_Tx( )*. For these, another external CAN node is required as transmitter or receiver station.

In the following, the major focus is on the description of the basic testing application.

5.5.3.1 General Approach

The following steps are performed in sequence in order to perform a basic test:

- Configuration
  - Definition of configuration (global and used channels)
  - Set up of required structures and data (messages - their configuration and storage)
  - Initialization of I/O Ports
  - Configuration loading and interrupt assignment
  - Message buffer or filter initialization

- Activation of operation mode

- Execution of communication function
  - Recognition of interrupts
  - Reading controller status
  - Retrieving messages on reception
  - Setting up messages and subsequent transmission

- Shutdown of the CAN controller

5.5.3.2 Basic Communication with AFCAN

This refers to implementations (A) and (B).

The functionality of this application is implemented in the function *<xxx>_BasicTest( )*. The function is using two different CAN controllers and channels, and includes an option to run for several passes without re-initializing.

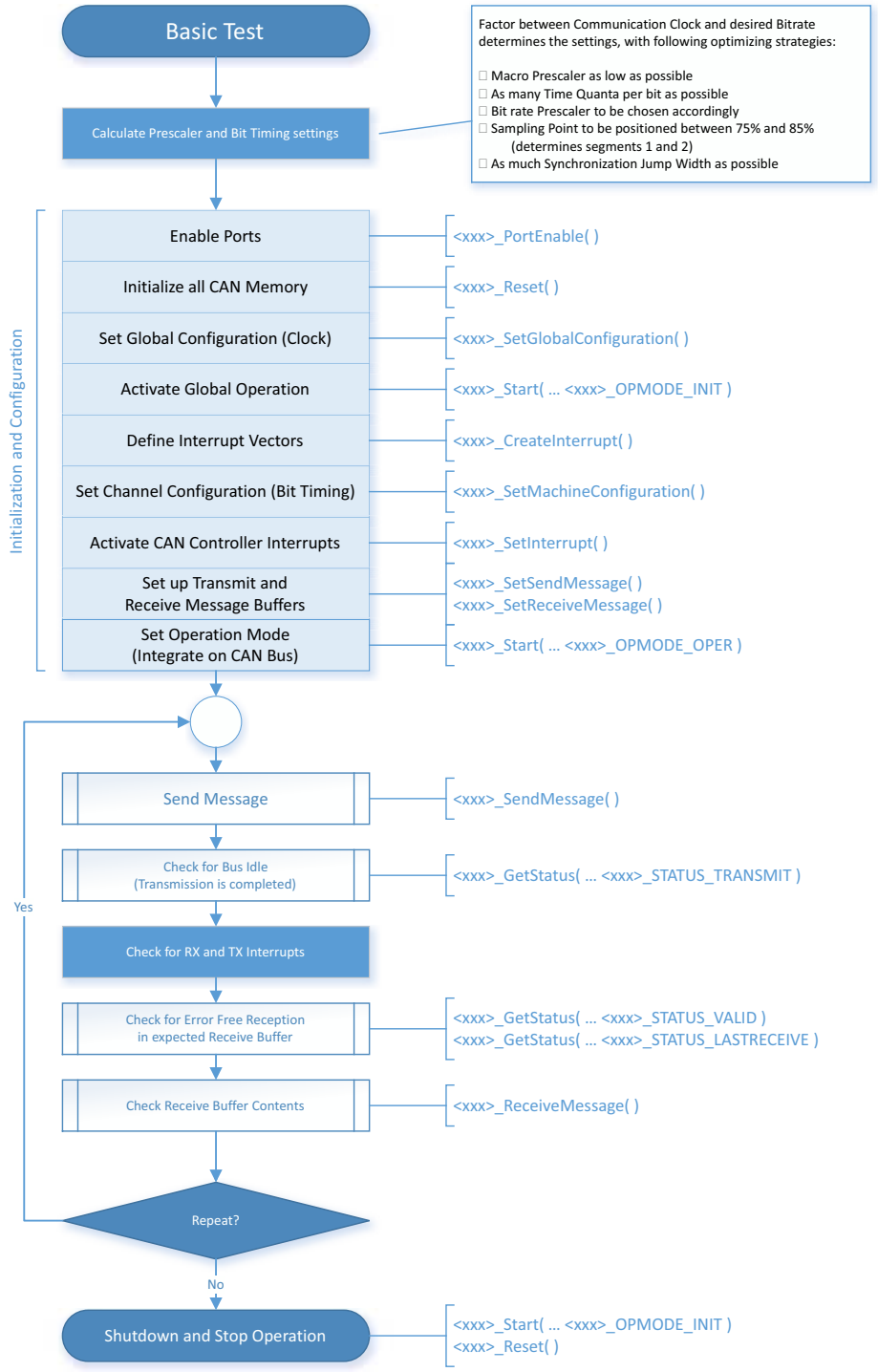

Figure 5.1 AFCAN Basic Communication Application Example

5.5.3.3 Basic Communication with RS-CANLite

This refers to implementation (C).

The functionality of this application is implemented in the functions *<xxx>_BasicTest_Tx( )* and *<xxx>_BasicTest_Rx( )*.

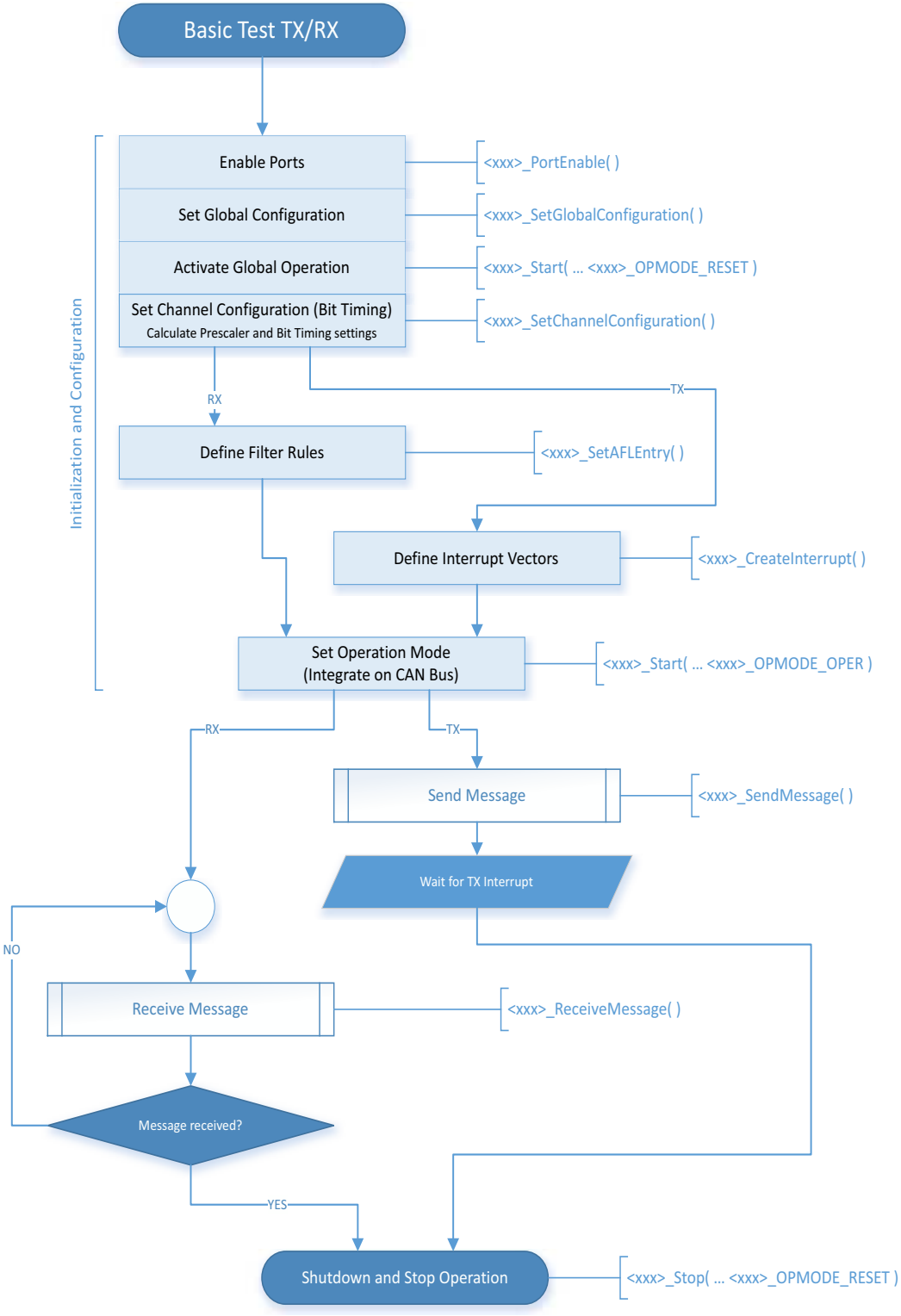

Figure 5.2 RS-CANLite Basic Communication Application Example

5.5.3.4 Basic Communication with RS-CAN(-FD)

This refers to implementations (D), (Ex), (E4).

The functionality of this application is implemented in the function *<xxx>_BasicTest( )*. The function is using two different CAN controllers and channels.

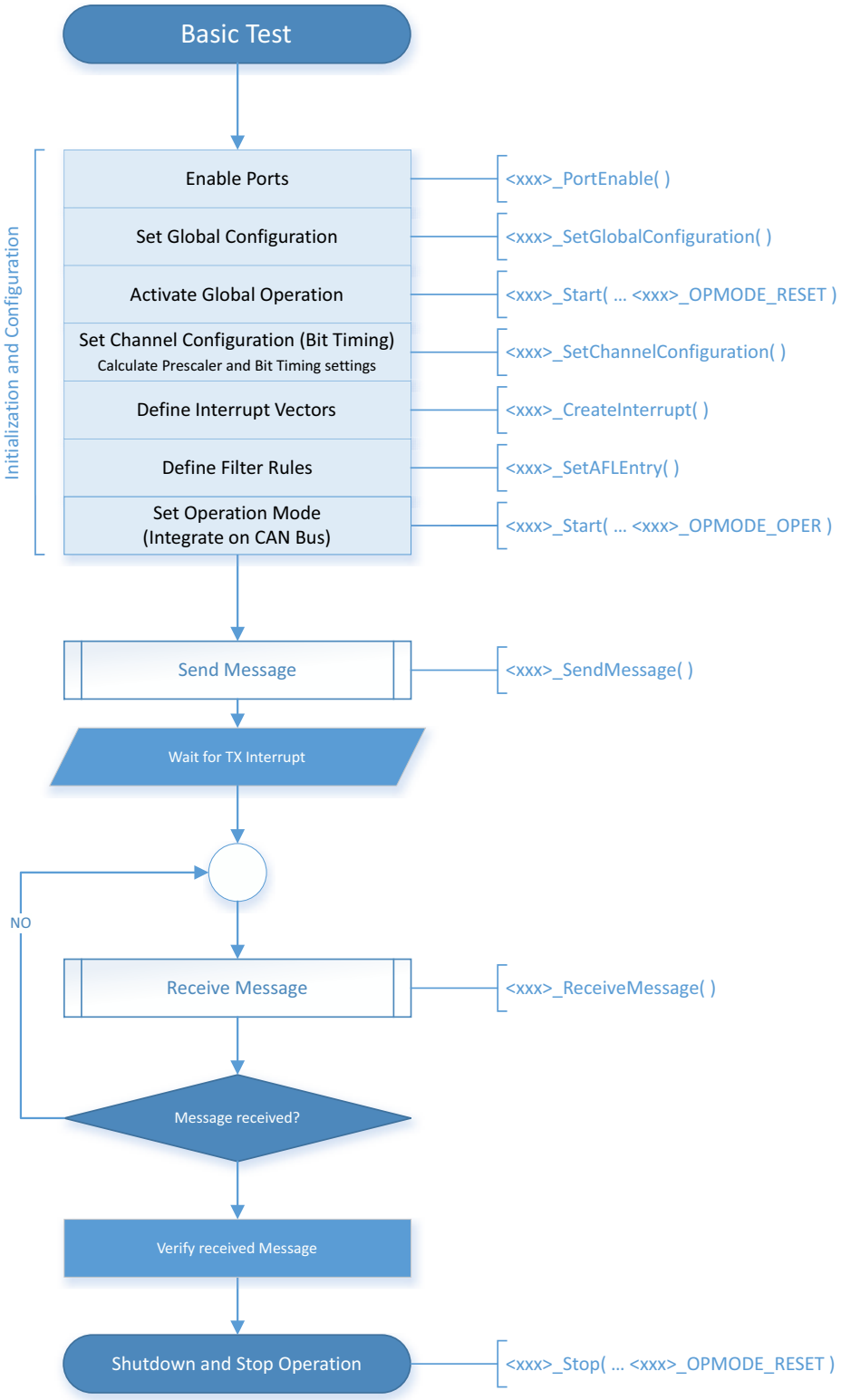

Figure 5.3 RS-CAN(FD) Basic Communication Application Example

5.5.3.5 Self Test with RS-CAN(FD)

This refers to implementations (D), (Ex), (E4).

The functionality of this application is implemented in the function *<xxx>_BasicSelfTest( )*. The function is using two different CAN controllers and channels.

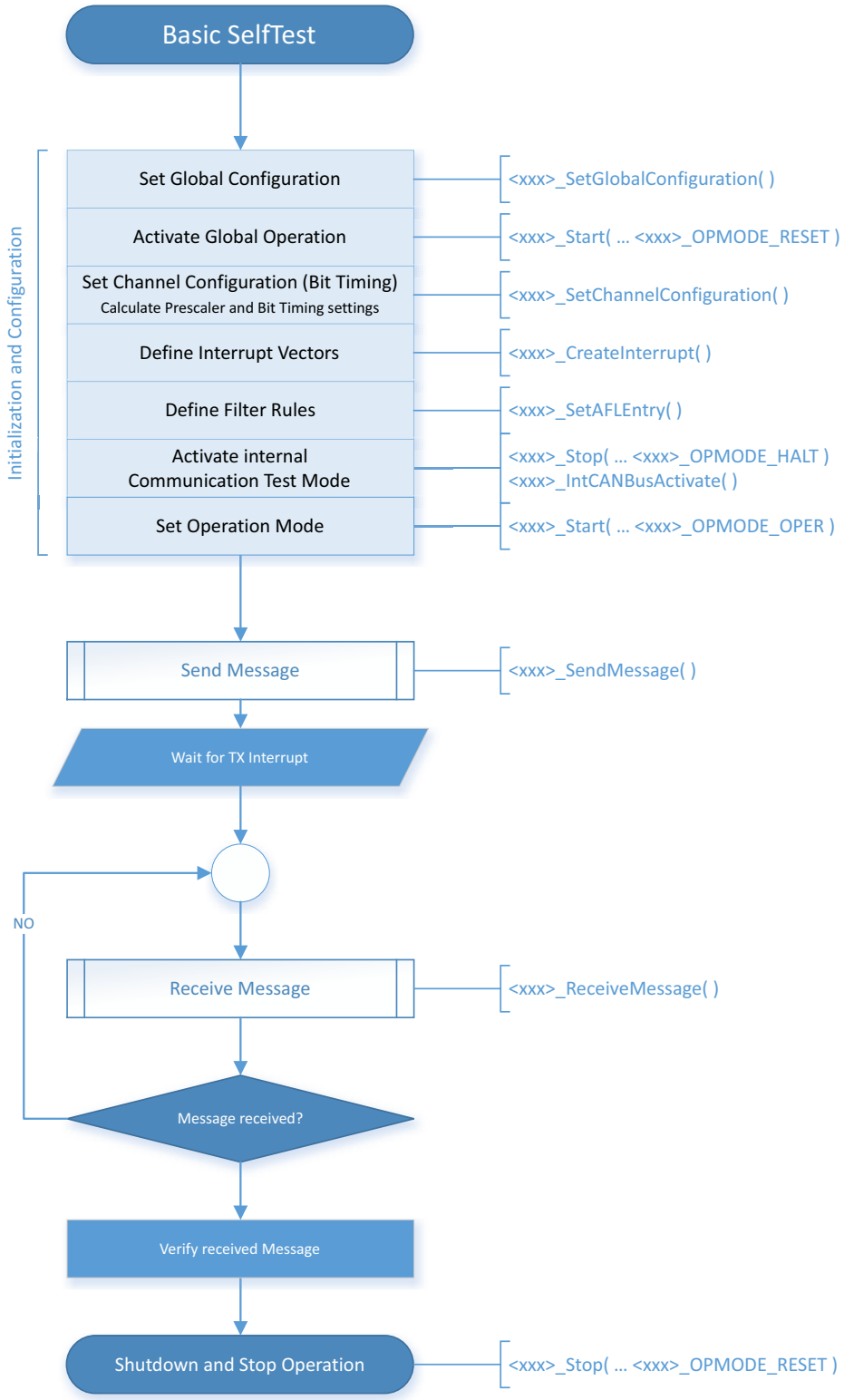

Figure 5.4 RS-CAN(FD) Self Test Application Example

5.5.3.6 Internal Self Test with RS-CAN(FD, -Lite)

This refers to implementations (C), (D), (Ex), (E4).

The functionality of this application is implemented in the function *<xxx>_BasicIntSelfTest( )*. The function is using one single CAN controllers or channel. It allows a simple hardware test which can efficiently applied to a single channel, i.e., for functional safety applications.

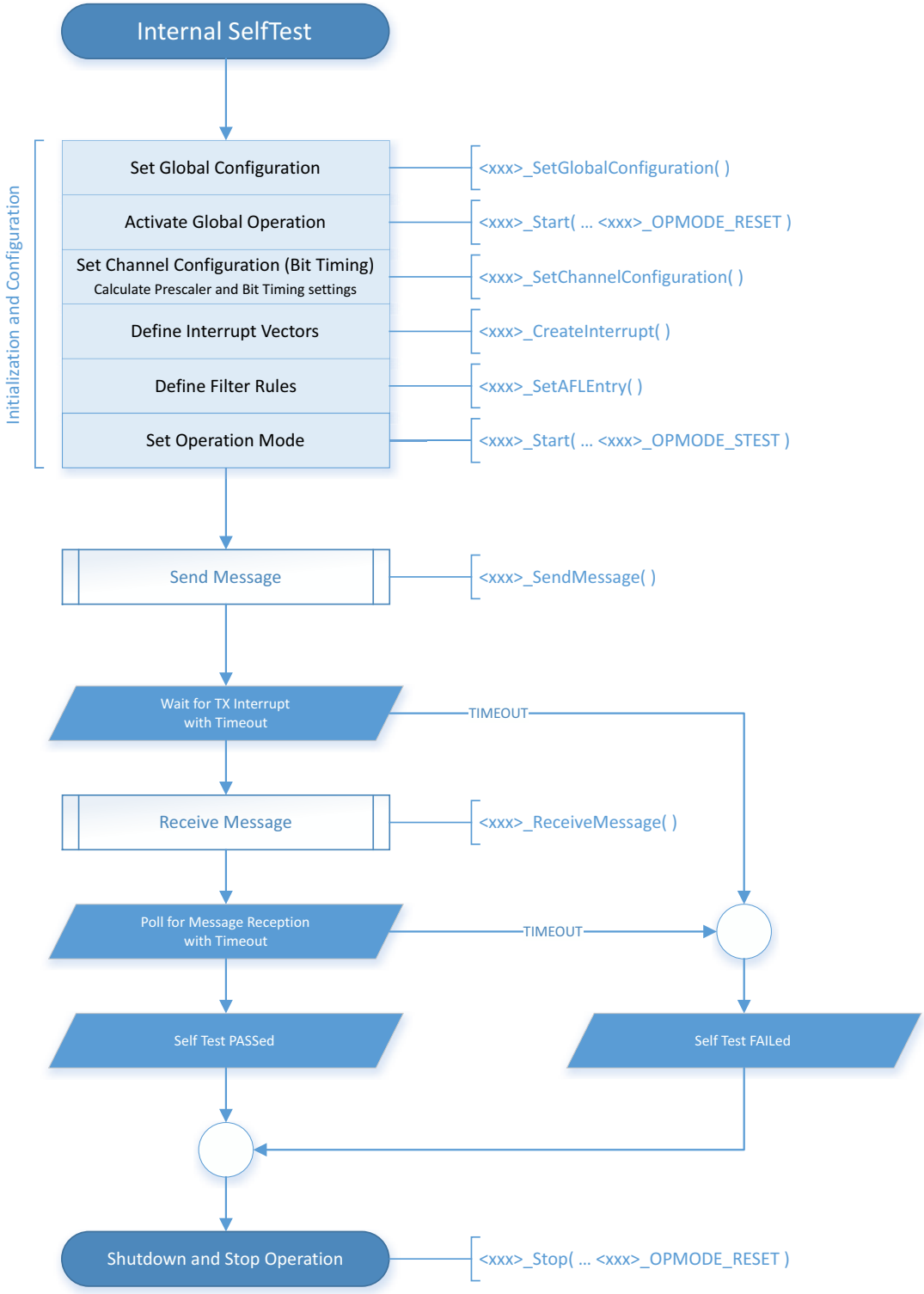

Figure 5.5 RS-CAN (FD, -Lite) Internal Self-Test Example

5.5.3.7 Basic Communication with M_(TT)CAN

This refers to implementation (F), (G).

The functionality of this application is implemented in the function *<xxx>_BasicTest( )*. The function is using two different CAN controllers.

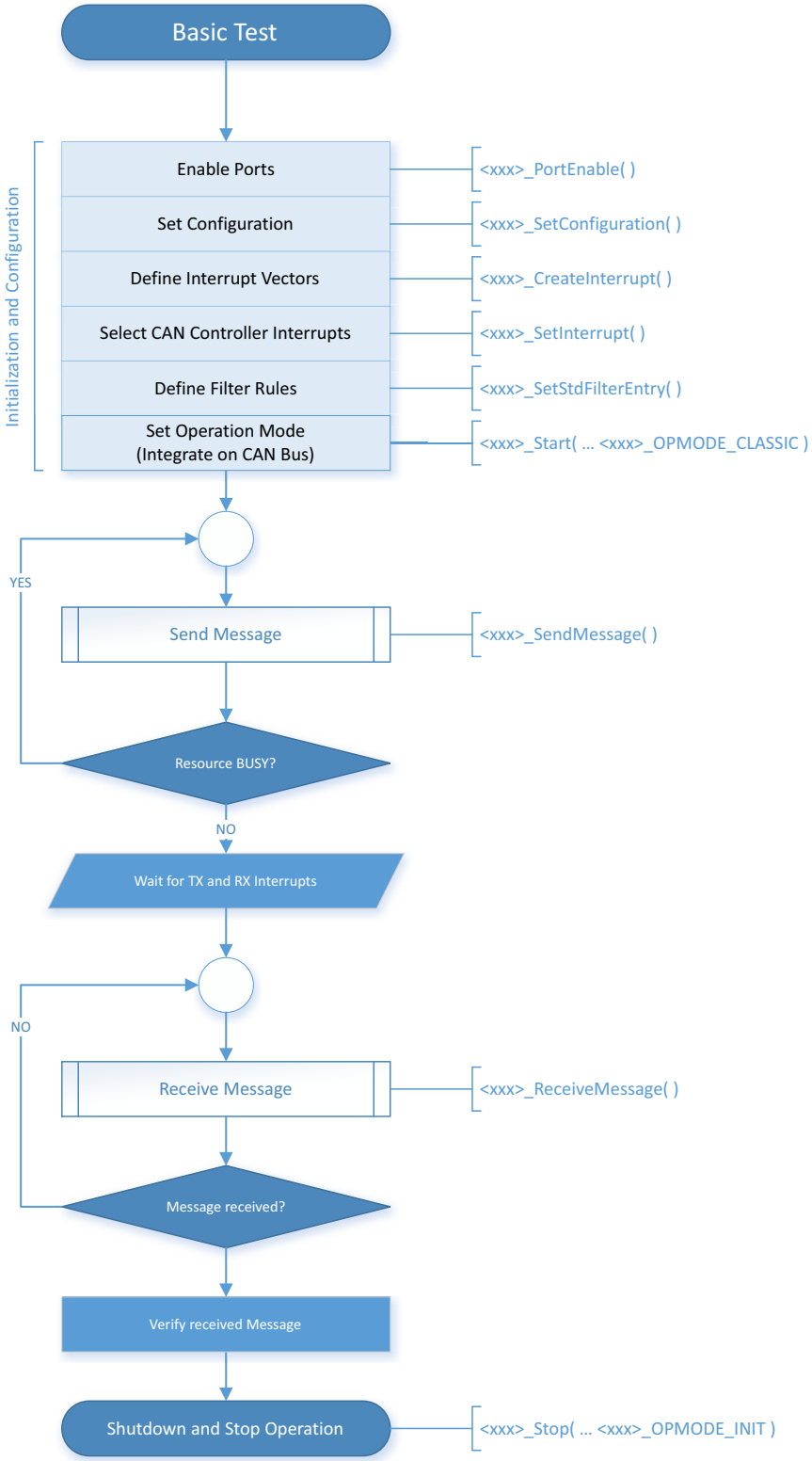

Figure 5.6 M_(TT)CAN Basic Communication Application Example

5.5.3.8 Software-Gateway with M_(TT)CAN

This refers to implementation (F), (G).

The functionality of this application is implemented in the function *<xxx>_Gateway( )*. The function is using at least two (or more) different CAN controllers. All standard identifier messages are copied from the received channel to all others. An exit condition is set by reception of any message having an extended identifier.

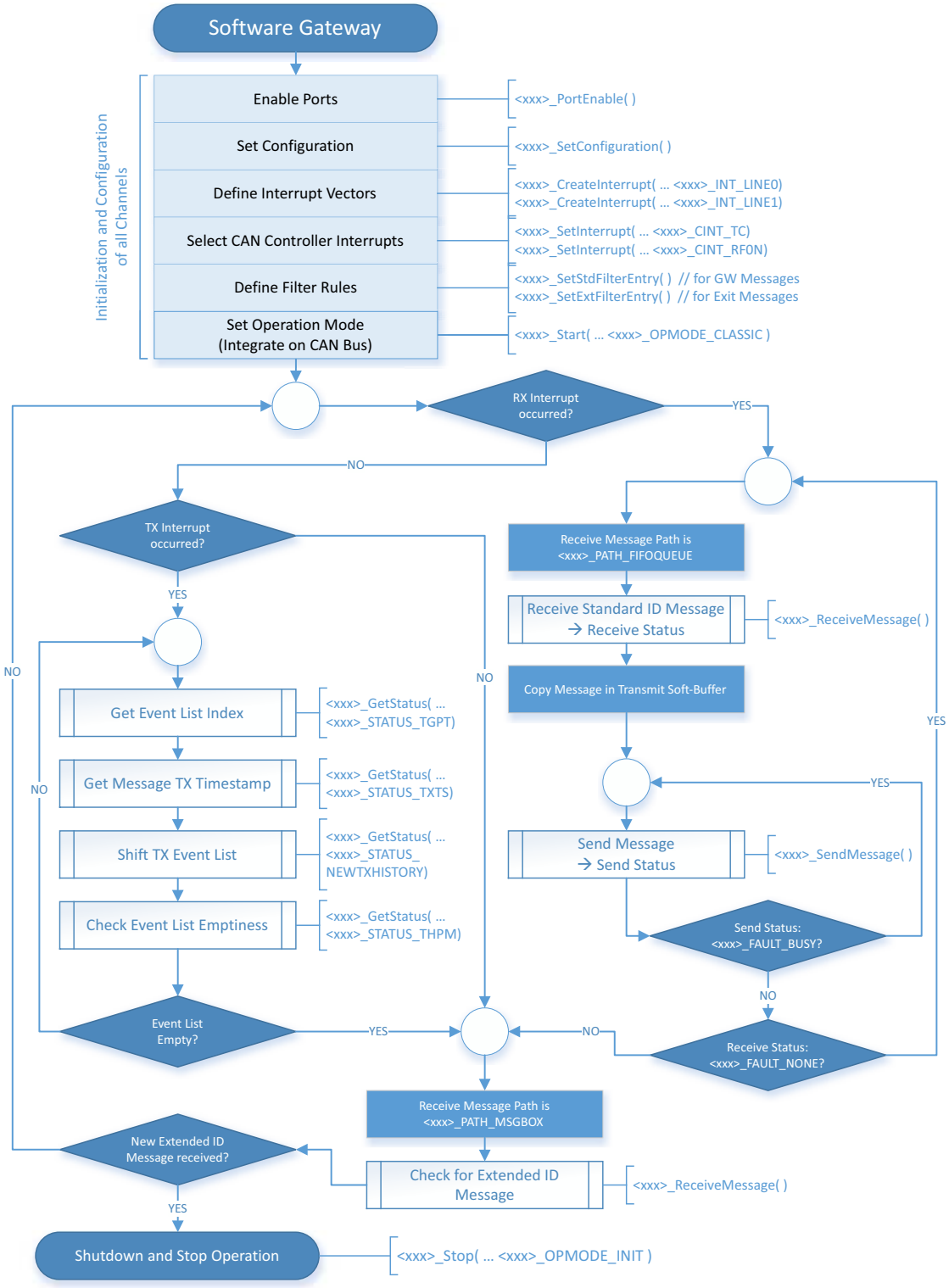

Figure 5.7 M_(TT)CAN Soft-Gateway Application Example

6. Frequently Asked Questions

6.1 CAN Conformance and Licensing

6.1.1 CAN Conformance Policy of Renesas

CAN Conformance Certificates according to ISO 16845 are applied commonly to several products, if all of the following conditions are true:

(1) Microcontroller devices (MCU) are belonging to the same family
(Example: RH850/D1x family, x can be replaced by the letter of the family member).

(2) Microcontroller devices (MCU) are having the same manufacturing technology for the CAN IP
(Example: same lambda-size of softmacro synthesis mapping, but different flash memory technology)

(3) Microcontroller devices (MCU) are containing the same CAN controller IP type and version
(Example: CAN IP type “RS-CAN” version 1.0)

6.1.2 CAN Conformance Test Specification

For any device which is released before 2015, the applied CAN Conformance Test Specification is ISO 16845:2004. After 2015, CAN Conformance Test Specification ISO 16845:2015 is applied.

6.1.3 Certification on ISO 17025

The test system of C&S, which Renesas is using and licensing, is according to ISO 17025. However: the Renesas test location is not directly touched by ISO 17025 - see the statements of ISO.

6.1.4 Proving of the CAN(-FD) License of Renesas CAN Controllers

Renesas has purchased a license from BOSCH. The license number is: L-107980. This license is valid for both CAN classic and FD applications. More information on this issue can be directly asked from BOSCH.

6.1.5 Extended Identifiers and SAE J1939

RS-CANFD, RS-CAN, RCAN, FCN, DCN, AFCAN and DAFCAN Macros are supporting SAE J1939, because of usage of Extended ID. The DCAN controller is not supporting extended identifiers.

6.1.6 Remote Frames

Remote frames are supported by all Renesas CAN controller types.

6.2 Transceiver Issues

6.2.1 Necessity of a CAN Transceiver

There are many ways to communicate the serial data of CAN. It is a requirement of the CAN controller, that it must be able to “see itself” sending, so there must be a functional feedback between the TX and RX lines. Such can be achieved simply by shorting these lines.
However, to match the requested physical layer of an ISO specification (i.e., 11898-2), the CAN transceiver is mandatory. Renesas devices are mostly not including a CAN transceiver.

6.2.2 Only SOF bit can be seen on the CAN bus

This happens, if the CAN transceiver is not operating as expected, and does not read back on its RX output, what it has got for sending to its TX input from the CAN controller. Check that always RX==TX, if TX=0. If this is not the case, either the transceiver is switched off, not powered, or broken.

6.2.3 Usage of the SPLIT Terminal

The SPLIT terminal of the transceiver can be connected to the middle of 60+60 OHMs, which is the bus termination resistor. If this is not convenient, leave SPLIT open.
SPLIT stabilizes the bus against common-mode voltage steps on partly powered networks, thus reducing EME.

6.3 Bit Timing and Clock Jitter

6.3.1 Calculating total Bit Timing Deviation with Jitter

The PLL Jitter is not accumulative for tolerance, because it is a random function. It adds once as an absolute value in worst case. For this reason, the long-term jitter of the used PLL is required for this calculation. The jitter measurement period should match the bit length of the used bit rate.

6.3.2 Usage of a PLL as a Clock Source for CAN

The quality of the PLL of Renesas devices regarding jitter and stability of the clock is generally good enough to allow the usage of the PLL as a clock source for the CAN controllers. All experiences up to now have never shown up any problem.
Even for GMLAN, it is not forbidden to use a PLL as communication clock source.
However, if a ceramic resonator is attached to the oscillator instead of a quartz, the deviation of the resonator in conjunction with the jitter may lead to a violation of the timing requirements. Therefore, we recommend the usage of the oscillator clock directly, if a resonator is used instead of a quartz.

6.3.3 Sporadically shortened or lengthened Bits by one or several TQ

Soft synchronization is shortening or lengthening the bits during frame transmission.
In the arbitration and acknowledge phase, where two or several participants are synchronizing with each other, the effect can be seen on the bus.
Therefore, this is a normal behavior, which is according to the CAN specification.

6.3.4 Drive Strength of Microcontroller I/O Port for CAN

The drive strength of the I/O port of the microcontroller does have an influence on the CAN bus bit timing accuracy. To ensure the Bit Timing quality, the Port Drive Strength should be at its maximum value, if the CAN Baudrate is higher than 200 kbit/s.

6.3.5 Bit Sampling Methods

After first hard synchronization on SOF, all bits of a classical CAN frame are handled equally, with a sampling at the sampling point and using soft synchronization by SJW.
While the CAN Transfer Layer (TL) used in AFCAN, FCN, RCAN, RS-CAN and RS-CANFD samples only once, the TL used in FCAN and DCAN samples three times, using a majority decision.

6.3.6 Resynchronization after a recessive to dominant edge in the SOF bit

Renesas CAN controllers are not performing a resynchronization, if a recessive to dominant edge occurs in the SOF bit before the sample point. This is according to ISO 11898-1.

6.3.7 Resynchronization outside of the SJW Range

Re-synchronization is performed on recessive to dominant edges even if they occur outside of the maximum SJW range. Since the maximum SJW is not sufficient in this case, the synchronization range is limited to SJW. This functionality is according to the CAN protocol, ISO 11898-1.

6.3.8 Information Processing Time (IPT)

Frominternal processing speed, the IPT of AFCAN, FCN, RCAN, RS-CAN and RS-CANFD is zero. However, as the minimum setting for the TSEG2 segment of the bit timing is 1, the effective IPT is equal to 1, and cannot be reduced further.

6.3.9 Port Initialization to avoid Spikes on the CAN bus

The following sequence is recommended: First Port Mode Control (select peripheral for the port - PMC and PFC(E)) to assign the CAN controller, then Port Mode (input or output mode - PM) to activate the output. If the port is a wired-x function with the value of the port register P, this register has to be initialized also to its non-penetrating value, before setting PM.

6.3.10 Bit Timing on CAN-FD Setting Recommendations

In order to achieve proper and safe communication with CAN-FD, some rules should be followed, which are also based on recommendations of CiA (CAN in Automation).

(1) Keep exactly the same sampling point for all nodes within your network.
This is valid for both arbitration and data phase bit timing settings.
(2) In order to achieve the previous recommendation, use the same oscillator clock speeds for all nodes.
It is recommended to use only the dedicated clock speeds of 20 MHz, 40 MHz or 80 MHz as communication clock source for the CAN-FD controller.
(3) Set the sampling point and the secondary sampling point positions to their optimum position regarding phase reserve for arbitration and data phase bit timing. Details are given in the *CAN in Automation (CiA)* publications 601.1 to 601.4.
(4) For the best start, set the SJW setting to its allowed maximum for your bit timing settings.
This allows the nodes to synchronize with widest range and get more tolerant against phase shiftings by that.
(5) Do not use a speed ratio larger than 10 between data and arbitration phase bit rates.
Higher ratios are causing that the transitions between arbitration and data phases (and vice versa) are getting more critical regarding proper sampling of the bits at the speed transition points.

As typical example of failure, see the following figure, where the first recommendation from above was not followed. It is obvious, that in this case, the bit rate switch (BRS) will fail, even though the CAN-FD protocol without BRS would work fine.

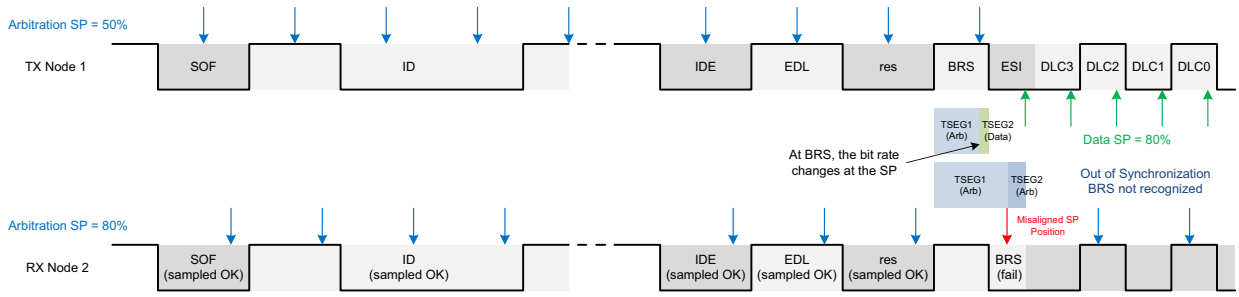

Figure 6.1 BRS Misinterpretation due to different Sampling Point Positions

The different sampling point of the arbitration bit rate causes, that the bit BRS is not sampled correctly, because at BRS, the bit rate switches to the data bit rate just at the sampling point.
This situation shows, that frames without bit rate switch (BRS) are received correctly by node 2, but as soon as a bit rate switch happens, node 2 runs out of synchronization and will destroy the frame.

While considering the facts as above, the following is allowed nevertheless:

- When only classical CAN frames are used in CAN-FD operation modes, it is recommended to set the bit timing parameters of arbitration and data phases such, that they result in the same bit rate, even if the data bit rate would never be used in this case.
Therefore, the data bit timing register should be set with values, that result in the same bit rate as defined with the arbitration bit timing register settings. Due to different resolution of values, a different combination of parameters (i.e., prescaler, data bit time) is allowed.

- It is recommended to have the same value for both data bit rate prescaler and arbitration bit rate prescaler, because this avoids synchronization problems between CAN nodes. According to ISO 11898-1, this is not mandatory, however. As written above, identical sample point positions are sufficient. A different resolution of TQ per bit between CAN-FD nodes is a less critical issue, but may become significant in extreme non-linear network structures, where the range of synchronization needs to be fully usable with equal resolution.

6.4 Operation Modes and Initialization

6.4.1 Delay when entering Initialization / Halt Mode

The delay to set INIT or HALT mode depends on Reception/Transmission of messages, while the Transfer Layer is running. The initialization or HALT mode can only be reached, while no more transmission is triggered and the CAN bus is in intermission state.

6.4.2 Interrupting Bus Off Processing by Software (AFCAN)

Interrupting the recovery by switching back to INIT mode and again to any operational mode restarts the recovery with full length. This can lead to an endless situation, if the software always interrupts the recovery. The recovery procedure is not applicable, if a regular start of the system is performed, without any bus-off condition in advance.

6.4.3 Integration State

When setting the operation mode to start communication, a CAN controller always enters “integration state” first. In this state, according to ISO 11898-1, it has to wait for 11 subsequent recessive bits, until it may start any reception or transmission. It is not required for the user to wait until integration state has left; transmissions may already be triggered within integration state, even though they will be delayed.
Due to the condition of ISO 11898-1, on a 100% bus load condition, the integration state may take an endless time.
RS-CAN and RS-CANFD controllers are allowing to check the integration state. The integration state has been successfully passed, if the COMSTS flag is set. In order to check the transition to the operation mode, a check of COMSTS is not helpful, instead the CRSTSTS flag of the CAN channel can be checked.

6.4.4 Allowed Options in Operation Modes (RS-CAN, RS-CANFD)

RS-CAN and RS-CANFD controllers do have global and channel based operation modes. Certain actions of software are either allowed or forbidden regarding these modes. The following table is providing an overview.

Table 6.1 Allowances in Operation Modes[a]

Software Action	Global Mode			Channel Mode		
	RESET	HALT	OPERATION	RESET	HALT	OPERATION
Set Channel Modes	√			√		
		√		√	√	
			√	√	√	√
Define AFL rule memory (*RNC* settings)	√			√		
Define/change AFL rule	√	√	√	√	√	
Configure THL, FIFO or TX Queue (depth, interrupt mode, operation mode, TX buffer link)	√			√		
Enable THL, FIFO or TX Queue		√	√		√	√
Enable/disable FIFO or TX Queue interrupt	√	√	√	√	√	√
FIFO data read/write, shifting		√	√	√	√	√
TX Queue data write/shift TX Buffer trigger			√		√	√
Receive / Transmit messages			√			√
Bus Off recovery			√		√	
Perform RAM Test		√		√	√	

a. “√” indicates the allowance in the corresponding operation mode. Several marked modes within a mode class (global / channel) are indicating an *‘or’* condition. Global mode and channel mode combinations are always *‘and’* conditions.
Examples: Changing an AFL rule is allowed in either global mode, if the channel mode is not set to operation mode. In global HALT mode, only RESET and HALT modes of a channel are possible.

6.5 Power Save Modes

6.5.1 Re-Initialization when in Power Save Mode (AFCAN, FCN)

The re-initialization is only possible, if the power save mode is canceled first.
The correct sequence is:
(1) Clear SLEEP and STOP mode,
(2) Set INIT mode,
(3) (Optionally) perform a soft reset.

If the sequence fails (supervise by a timeout of at least 2 CAN frames in length, perform a forced soft reset:
(4) Set EFSD,
(5) Clear GOM.
(6) Set GOM.

6.5.2 Unconditional Wake-Up by any CAN Bus Event

The SLEEP mode is left automatically, if any dominant edge on the CAN bus is detected.
This mode is only supported by DCAN, AFCAN, and FCN controllers.
RCAN, RS-CAN and RS-CANFD controllers are not supporting this kind of wake-up detection. The available STOP mode does not wake up upon CAN bus activity. Here, an additional port-edge detection interrupt is required to perform unconditional wake-up.

6.5.3 Selective Wake-Up by a dedicated Identifier

This can only be performed, if the CAN controller is left in an operational mode (communication mode or receive-only mode), so that is can monitor the CAN bus and watch the incoming messages. Power saving modes (SLEEP, STOP) of the CAN controller are not suitable for this application.

6.5.4 Receive and Transmit Interrupts in SLEEP Mode

When entering SLEEP mode of DCAN, AFCAN, and FCN, receive and transmit interrupts may still occur. To deal with this, look for the following Software Requirements:
(1) Check INTS Register while waiting for the CAN hardware acknowledging (read back) of SLEEP Mode.
(2) On entering Rx or Tx interrupt routines, check the MBON flag to verify that the CAN controller is not in SLEEP mode, before accessing history lists or message buffers.

6.5.5 Dominant blocked CAN bus while in SLEEP Mode

Applicable on DCAN, (D)AFCAN, FCN and DCN only. There are two different cases.
(1) The CAN controller was in Initialization Mode (offline), during the CAN bus was blocked on dominant level.
In this case, the CAN controller will enter the Sleep Mode, but a wake up by a CAN bus event (falling edge) will happen only, if the dominant bus blocking is released and at least 11 recessive bits have been detected on the CAN bus, before the next falling edge (SOF) appears.
The reason for this is, that the CAN controller first must have the opportunity to synchronize on the bus, before the Sleep Mode can be processed completely. For the initial synchronization process, 11 recessive bits are required on the bus, if the CAN controller synchronizes from offline to online operation.
(2) The CAN controller was in operation (online), during the CAN bus was blocked on dominant level.
In this case, the Sleep Mode cannot be entered.

6.5.6 Preconditions for DAFCAN and DCN to enter a Power Save Mode

To enter Sleep/Stop Modes, both parts of DAFCAN/DCN must be put into this mode, i.e., the DIAG channel and the RXONLY channel.

6.6 Transmission

6.6.1 Transmit Abortion in general

It is not possible under normal circumstances, to abort the transmission of a message, where the transmission is already started and the transmission is already advanced such, that the arbitration has already taken place. Abortion of a message in this state is also not allowed from CAN protocol side (ISO 11898-1).
However, after each loss of arbitration, abortion is possible.
By exception, abort of transmission is also possible by forcing a shutdown of the CAN Controller. This will result in a violation of the CAN protocol (ISO 11898-1) and therefore cause a CAN bus error. After such a shutdown, the CAN controller must be re-initialized completely, to resume operation.

6.6.2 Transmit Abortion in (D)AFCAN, FCN, DCN

Transmission Abort is performed by clearing the TRQ Flag of an affected message buffer by software.
The abortion will be successful immediately, if the buffer is not yet being sent (waiting to be transmitted).
The abortion will be successful, if the sending has already started, but the arbitration of the sent frame is lost. The abortion will take place right after this.
The abortion will be successful, if the sending has already started, but during transmission, a bus error occurs. The abortion will take place right after this.
The abortion will be successful, if the sending has already started, but the frame is not acknowledged. The abortion will take place right after the negative acknowledge.
In all other cases, the abortion will be unsuccessful.

6.6.3 Transmission Confirmation with FIFO in RS-CAN, RS-CANFD

Mechanisms to provide confirmation of successful transmission with Common FIFO Buffers of RS-CAN and RS-CANFD:

- Checking the transmit history list (THL) by polling
  This method is limited by the size of the THL. If the FIFO is sending more messages than the THL can store, the THL will overflow. This method will work, if the FIFO size is limited to the THL size, if no other transmissions are performed apart from the FIFO, if the THL is polled as soon as the FIFO has become empty, and if the FIFO gets refilled earliest after that.
- Checking the transmit history list (THL) by interrupt
  It is required to enable the FIFO interrupt such, that the interrupt occurs on every successful transmission. This method allows the combination of several FIFO units and conventional transmit buffers, too. The capability to monitor each transmission success has the drawback that is will cause a certain interrupt load to the CPU.

6.7 Reception

6.7.1 Mixed Reception of Extended and Standard Frames in one Message Box

Masking of IDE is not possible in DCAN, (D)AFCAN, FCN and DCN.
Masking of IDE is possible in FCAN, RCAN, RS-CAN and RS-CANFD.

6.7.2 Masking (DCAN, FCAN, (D)AFCAN, FCN, DCN, RCAN)

These CAN controllers are supporting the ID masking principle.
Masks are checked sequentially for any incoming message. If the first matching mask and ID does not find a free message buffer, the message is rejected. Other masks are not checked again, if one mask has matched already, even though at this match, no free buffer should have been available.

6.7.3 Filtering (RS-CAN, RS-CANFD)

Instead of masking, these CAN controllers are using filtering lists (AFL) in the reception processing.

6.7.3.1 No Priority of Reception among Channels

If there are several channels, which are receiving messages into the same RX-FIFO, there is no priority among them. There is a priority among AFL rules (first rule is checked first), but if several incoming messages from different channels have identical reception targets by independent rules, then there is no given priority.
A central engine is scanning all channels in a loop, and if a channel has set a reception indication, it is served and its message gets processed. Therefore, the result about which channel receives first is random, depending on the current state of the central engine.
Of course, one could now ask: ‘what amount of time must I be in advance for a message on a channel, to be sure that I'm the first to be received?’. But at the same time, let’s keep in mind that CAN is not a synchronous protocol; so you would never know exactly when your message gets sent or received.
Except for TTCAN, there is no specification for that. But even for TTCAN, the specification is valid for one channel only, not for several channels at a time.

6.7.3.2 Masking the IDE Flag (Extended Frames) and ID Comparison

RS-CAN and RS-CANFD controllers are able to mask the IDE flag of incoming messages within an AFL rule. In order to do this, the *GAFLIDEM* flag has to be used.
Refining the descriptions in the manuals, the function of *GAFLIDEM* and *GAFLIDM* is as follows:

(1) *GAFLIDEM* within the AFL rule is set (1):
The rule applies only to messages, where IDE matches the setting of *GAFLIDE*.

(2) *GAFLIDEM* within the AFL rule is not set (0):
The rule applies to messages, where IDE is either set or not set (standard and extended messages).

In any case, depending on the received IDE of a message, the ID of the message is compared with *GAFLID* of the AFL rules.
- If IDE of the message is set, all bits of the ID are compared with *GAFLID,* where corresponding *GAFLIDM* flags are set.
- If IDE of the message is not set, the bits of the ID are compared with the lowest 11 bits of *GAFLID*, where corresponding *GAFLIDM* flags are set.

If a rule excludes an incoming message, because *GAFLIDEM* is set and IDE of the incoming message is not matching with *GAFLIDE*, then the rule is skipped and not applied.

6.7.3.3 Order of Reception Rules, Rule Count (RNC)

If the CAN controller unit has several channels, then for each channel the reception rules have to be defined. Within the AFL rules list, the rules must be ordered such, that all rules of channel 0 are entered first (on the first page), followed then by the rules of channel 1, channel 2 and so on. It is mandatory that no gaps between the rules are left, and the amount of rules as specified with the RNC*n* register is matching the amount of rules for that channel *n*. All rules are ordered with highest priority first for and within each channel.

6.8 Message Storage

6.8.1 Buffers (Mailboxes): FCAN, (D)AFCAN, FCN, DCN

6.8.1.1 Effect of the RDY Flag

If RDY is cleared, no message can be stored in the buffer; and no message can be sent from the buffer. CPU access to the buffer remains active, however.

6.8.1.2 Multi-Buffer Receive Blocks (MBRB) and Overwriting (OWS)

MBRBstill works, if OWS of a buffer is set, but as soon as all buffers of the MBRB are filled (DN=1), the first buffer of the MBRB will be overwritten. Other buffers of the MBRB can never be overwritten, because always the first one is used. The flag MOW indicates the overwriting, but it is not used as a qualifier to select other buffers for overwriting.

6.9 History Lists: (D)AFCAN, FCN, DCN

6.9.1 Handling after Overflow

After an overflow, the history list function is suspended, but old entries of the history list can be read and the associated message buffers can be processed.
However, there may be more message buffers, that need processing. If in case of reception, if the OWS flag of unprocessed message buffers is set, they will not be blocked for new reception, and processing of them can be omitted. If there are receive message buffers, where OWS is not set, the application must scan all these message buffers and clear the DN flag, in order to avoid blocking of these. After all this processing has been completed, the overflow flag of the history list shall be cleared, and all entries of the history list shall be read out, in order to re-establish the history list functionality.

6.9.2 Overwriting (OWS) Enable and Receive History List

Even if a buffer is overwritten, because its OWS Flag is set, this event is also stored in the receive history list.
This is very important to know; it is not possible to leave a buffer with OWS set unattended, because
its reception events could cause the RHL to overflow, if they are not handled.
In FCNand DCN, the “no history” flag (NH) can be used to suppress RHL entries of buffers; here, its usage in combination with OWS is recommendable.

6.9.3 Lost Receptions by wrong handling of RHL Registers

Lost message receptions at high CAN bus load can be caused by double reading the history list content within the receive processing loop, hereby discarding history list entries accidentally.
The correct processing is to read the history list only once per processing a message buffer.
To do so, read the history list register into a temporary variable, and use the temporary variable for further processing of the value, instead of reading the history list register again.
By every reading of the history list register, the history list pointer (and entry) is moved and updated.
Therefore, when reading the history list a second time while processing one message buffer, the entry of the next message buffer is already taken away from the list. Such processing causes that there will remain unprocessed message buffers, which then can remain blocked for further reception. This, however, can result in lost message receptions for the upper layers of the application software.

6.10 Peripheral Bus Access

6.10.1 32-Bit Accesses

At (D)AFCAN not allowed in general, but by exception CnMDATA registers on longword-aligned addresses can be accessed in 32-bit accesses.
RS-CAN, RS-CANFD, FCAN, FCN and DCN controllers have dedicated registers for 32-bit access size.
RCAN and DCAN are not supporting 32-bit accesses.

6.10.2 Minimum Peripheral Bus Clock Speed

The RS-CAN and RS-CANFD controllers are supporting separate clock specifications for register access (“peripheral bus clock”) and communication.
In order to have proper processing of reception and transmission, the peripheral bus clock shall be at least double in speed as the communication clock. If this is not followed, sporadic losses of received frames or failed transmissions may occur.

6.11 Interrupts

6.11.1 Lost Interrupts in (D)AFCAN, FCN, DCN when disabling by CxIE or CnMCTRLm.IE

CxIE and CnMCTRLm.IE are not storing pending interrupts, while the interrupt controller of the device (using the interrupt controller registers) does it. Therefore, to avoid lost interrupts on temporarily disabled interrupt, always use the interrupt controller registers of the interrupt controller; not the CAN controller registers CxIE or CnMCTRLm.IE.

6.11.2 Conditions in (D)AFCAN, FCN and DCN regarding CINTSx and CIEx

Precondition: MCTRLm.IE is set.

- If CIEx is not set, while CINTSx is set by an interrupt condition, no interrupt is forwarded to the interrupt controller.
- If CIEx is set, after CINTSx is already set, no interrupt is forwarded to the interrupt controller.
  Only if CIEx is set, and after or during this, CINTSx is set by an interrupt condition, an interrupt is forwarded to the interrupt controller.

6.11.3 Missing Receive Interrupts ((D)AFCAN, FCN, DCN)

Thereare several reasons for this, however three most often seen are the following, which are not so easy to detect and described here therefore.

(1) Blocked Message Boxes
A blocking of further interrupts can occur, if message boxes are no longer available to receive messages, because they are already holding messages. This happens, if the CPU is not clearing the DN flag and overwrite mode for a message box is not enabled.

(2) Blocked Message Boxes in conjunction with AUTOSAR functionality
Regarding AUTOSAR functionality, there is a potential danger to get blocked message boxes, whenever a double mode change of the CAN controller is requested. Whenever the CAN controller has to go to initialization mode, a last reception may occur, which sets DN of a message box. However, the re-transition to an operation mode also clears the Receive History List.
Like this, a following interrupt routine would be unable to find this message box, so that it remains blocked. The workaround is, that whenever initialization mode is reached, all DN flags of all receive message boxes must be cleared.

(3) Not enabled interrupt in CAN controller interrupt enable register
If the interrupt enable register (CxIE1) is not set, while a RX interrupt occurs, this interrupt gets lost. If, as a consequence, the corresponding message buffer DN flag is not cleared by the CPU, there will be no more further interrupts for this message box.

6.11.4 Suppression of Receive Interrupts for Remote Frames ((D)AFCAN)

Remote frames are received in enabled TX Buffers (RDY set), where the ID matches the ID of the remote frame. If the ID values of the remote frames are known, which shall be suppressed, declare dedicated TX buffers for them, and disable the IE flag for them. The buffers shall be used from #0 upwards, so that the transmit buffer for sending messages has a higher buffer number than those used to discard the remote frames. Like this, it cannot happen that in the sending buffer the remote frames are received, because accidentally, the ID to send matches one of the remote frames.
If the ID values of the remote frames are unknown and random, there is no real clue to avoid Rx interrupts from those. At any time, a declared Tx Buffer could match the ID of the remote frames, and then be used the receive the remote frame, causing a Rx interrupt. One could disable the Tx Buffers by clearing RDY, as soon as the transmission is completed, to avoid the storage of remote frames in them. But still, a short time frame may be left open, where a remote frame could jump in.

6.11.5 Interrupt Handling in RL78 RS-CANLite Implementations

The interrupt controller in RL78 is triggered by edges of interrupt indications of peripherals, and so for RS-CANLite, too. On the other hand, the interrupt sources of RS-CANLite are level based.
For this reason, when handling RS-CANLite interrupts in RL78, all interrupt sources within RS-CANLite, which are sharing the same interrupt flag of RL78 must be handled and cleared, as soon as the interrupt is executed by RL78.
As an overview, the following interrupt sources of RS-CANLite are grouped in RL78:

Table 6.2 Shared RL78 Interrupt Sources of RS-CANLite

Interrupt Source	Shared Interrupt Events	Interrupt Event Flags to Clear
Global RX FIFO Reception	Receive FIFO *m*	RFIF in the RFSTS*m* register
Global Error	DLC Error	DEF in the GERFL register
	Message Lost Error	RFMLT in the RFSTS*m* registers CFMLT in the CFSTS*k* registers
	THL Entry Lost Error	THLELT in the THLSTS register
Channel *n* Transmission	Transmission Complete	TMTRF set to 00B in all TX Message Buffers of Channel *n*
	Transmission Aborted	
	FIFO Transmission	CFTXIF in the CF*n*STS*k* registers
	Transmission History (THL)	THLIF in the THL*n*STS register
Channel *n* FIFO Reception	Receive FIFO *n*, *k*	CFRXIF in the CF*n*STS*k* registers
Channel *n* Error	Bus Error	BEF in the C*n*ERFL register
	Error Warning Level	EWF in the C*n*ERFL register
	Error Passive Level	EPF in the C*n*ERFL register
	Bus Off Entry	BOEF in the C*n*ERFL register
	Bus Off Recovery	BORF in the C*n*ERFL register
	Overload Flag	OVLF in the C*n*ERFL register
	Bus Lock	BLF in the C*n*ERFL register
	Arbitration Lost	ALF in the C*n*ERFL register

Note: As an example, in order to get another channel 0 transmission interrupt, in the associated interrupt routine of the channel 0 transmission interrupt source, all TMTRF flags of all TX message buffers need to be checked and cleared, if set; further, the CFTXIF flag of the common TX FIFO needs to be checked and cleared, if set; and finally, the THLIF flag of the transmit history list needs to be checked and cleared, if set.
Flags of other channels need not to be handled cleared, as these are covered by separate interrupt sources.

Website and Support

Renesas Electronics Website

http://www.renesas.com/

Inquiries

- http://www.renesas.com/contact/

Revision History	CAN Controller Application Note

Rev.	Date	Description	
		Chapter	Summary
01.00	January 2015	—	First Edition issued
01.01	February 2015	Index, 3, 4	Index added, FAQ extended, Pretended Networking Chapter added
02.00	July 2016	1, 2, 3, 5 3 4	New content added Moved to chapter 4 Moved to chapter 6
02.01	August 2017	6 all	Topics added Considering RS-CANFD V2 and V3 versions
02.02	Feb 2019	1 5 6 all	Memory layout of RS-CANFD V2 and V3 corrected. Function <xxx>_GetStatus() modified for [E3] implementation (supporting TS). RL78/F15 added as supported device family for the CAN sample software. Topics added Considering RS-CANFD V4 version
02.03	May 2019	1, 5	References for RH850/E2x devices corrected.

General Precautions in the Handling of MPU/MCU Products

The following usage notes are applicable to all MPU/MCU products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Handling of Unused Pins

   Handle unused pins in accordance with the directions given under Handling of Unused Pins in the manual.

   — The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible. Unused pins should be handled as described under Handling of Unused Pins in the manual.

2. Processing at Power-on

   The state of the product is undefined at the moment when power is supplied.

   — The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the moment when power is supplied.
   In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the moment when power is supplied until the reset process is completed.
   In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the moment when power is supplied until the power reaches the level at which resetting has been specified.

3. Prohibition of Access to Reserved Addresses

   Access to reserved addresses is prohibited.

   — The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.

4. Clock Signals

   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has stabilized.

   — When the clock signal is generated with an external resonator (or from an external oscillator) during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Moreover, when switching to a clock signal produced with an external resonator (or by an external oscillator) while program execution is in progress, wait until the target clock signal is stable.

5. Differences between Products

   Before changing from one product to another, i.e. to a product with a different part number, confirm that the change will not lead to problems.

   — The characteristics of an MPU or MCU in the same group but having a different part number may differ in terms of the internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
3. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from such alteration, modification, copy or otherwise misappropriation of Renesas Electronics product.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; and safety equipment etc.
   Renesas Electronics products are neither intended nor authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems, surgical implantations etc.), or may cause serious property damages (nuclear reactor control systems, military equipment etc.). You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application for which it is not intended. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for which the product is not intended by Renesas Electronics.
6. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You should not use Renesas Electronics products or technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. When exporting the Renesas Electronics products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, who distributes, disposes of, or otherwise places the product with a third party, to notify such third party in advance of the contents and conditions set forth in this document, Renesas Electronics assumes no responsibility for any losses incurred by you or third parties as a result of unauthorized use of Renesas Electronics products.
11. This document may not be reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

RENESAS

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2801 Scott Boulevard Santa Clara, CA 95050-2549, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
1101 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada
Tel: +1-905-898-5441, Fax: +1-905-898-3220

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100191, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, P. R. China 200333
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics Korea Co., Ltd.
12F., 234 Teheran-ro, Gangnam-Ku, Seoul, 135-920, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141