VECTOR

MICROSAR LIN Interface

Technical Reference

AUTOSAR 4

Version 4.04.00

Authors	Bastian Molkenthin
Status	Released

Document Information

History

Author	Date	Version	Remarks
Bastian Molkenthin	2012-07-16	1.00.00	Initial version
Bastian Molkenthin	2012-11-08	1.00.01	Corrected misspellings
Bastian Molkenthin	2013-03-05	1.01.00	Added Post-build support, removed unused Vector parameters
Bastian Molkenthin	2013-09-30	2.00.00	Improved several chapters, removed redundant description of BSWMD parameters, added Runtime measurement
Bastian Molkenthin	2014-01-07	2.01.00	Added AR4-569 changes, adapted architecture and TP timeout figures
Bastian Molkenthin	2014-05-05	3.00.00	Removed BRS upper layer parameter Adapted wakeup confirmation handling LinTp/PduR interface conform to AR4.1.2
Bastian Molkenthin	2014-09-16	4.00.00	Added Post-build Selectable, Load Balancing, ComStackLib optimizations. Wakeup handling compliance to AR4.1.x (LinIf_WakeupConfirmation), Broadcast request handling, LIN communication error detection
Bastian Molkenthin	2015-01-08	4.01.00	Added schedule end notification
Bastian Molkenthin	2015-06-18	4.02.00	Added diagnostic communication chapter
Bastian Molkenthin	2016-02-08	4.03.00	Added SAE J2602 frame tolerance support
Bastian Molkenthin	2017-01-19	4.04.00	Added multiple driver support Added schedule info API Added Lin driver status callout Added schedule table change point Added external wakeup delay

Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	AUTOSAR_SWS_LINInterface.pdf	4.0.0
[2]	AUTOSAR	AUTOSAR_SWS_DevelopmentErrorTracer.pdf	3.2.0
[3]	AUTOSAR	AUTOSAR_SWS_DiagnosticEventManager.pdf	4.2.0
[4]	AUTOSAR	AUTOSAR_TR_BSWModuleList.pdf	1.6.0
[5]	AUTOSAR	AUTOSAR_SWS_BSWModeManager.pdf	1.2.0
[6]	LIN	LIN Specification Package, Revision 2.1, November 24, 2006	2.1
[7]	MICROSAR	MICROSAR LIN State Manager Technical Reference	see delivery

© 2017 Vector Informatik GmbH

[8]	AUTOSAR	AUTOSAR_SWS_RTE.pdf	3.2.0
[9]	MICROSAR	MICROSAR PDU Router Technical Reference	see delivery
[10]	MICROSAR	MICROSAR RTE Technical Reference	see delivery
[11]	MICROSAR	MICROSAR ComStackLib Technical Reference	see delivery
[12]	MICROSAR	MICROSAR COM Technical Reference	see delivery
[13]	MICROSAR	MICROSAR Complex Device Driver	see delivery

Scope of the Document

This technical reference describes the general use of the MICROSAR LIN Interface and of the MICROSAR LIN Transport Protocol.

The term LINIF is not strictly used for the LIN Interface, but also for the LIN Transport Protocol.

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

Following icons are used within this document:

Caution
This symbol calls your attention to warnings.

Note
Here you can obtain supplemental information.

Practical Procedure
Step-by-step instructions provide assistance at these points.

Do not edit manually
This symbol warns you not to edit the specified file.

Edit
Instructions on editing files are found at these points.

© 2017 Vector Informatik GmbH

Contents

Illustrations

Tables

1. Component History

The component history gives an overview over the important milestones that are supported in the different versions of the component.

Component Version	New Features
1.00.00	First version of the module supporting AUTOSAR R4.0.3
1.02.00	Support of Post-Build-Loadable
2.00.00	Support of multiple LinTp connections to same LIN slave node Support of Runtime Measurement
3.00.00	Support of ComStackLib
4.00.00	Support of MICROSAR Identity Manager using Post-Build Selectable Support of Load Balancing Support of external wakeup handling according to AUTOSAR R4.1.2
4.01.00	Support of schedule end notification
5.01.00	Support of SAE J2602 frame tolerance support
5.02.00	Support of Multiple LIN driver handling Support of ScheduleInfo API

Table 1-1 Component history

2. Introduction

This document describes the functionality, API and configuration of the AUTOSAR BSW module LINIF as specified in [1]. Also the integration into the AUTOSAR stack is covered by this document.

Please note that in this document the term Application is not used strictly for the user software but also for any higher software layer, like e.g. the AUTOSAR Network Management Interface. Therefore, Application refers to any of the software components using the LIN interface.

Supported AUTOSAR Release*:	4	
Supported Configuration Variants:	PRE-COMPILE [SELECTABLE] POST-BUILD-LOADABLE [SELECTABLE]	
Vendor ID:	LINIF_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	LINIF_MODULE_ID	62 decimal (according to ref. [4])

* For the precise AUTOSAR Release 4.x please see the release specific documentation.

The MICROSAR LIN Interface provides an abstraction layer for the used LIN Drivers. It offers an interface to the upper layer components.

2.1 Naming Conventions

The names of the service function provided by the LINIF always start with a prefix that denominates the software component where the service is located. E.g. a service that starts with '`LinIf_`' is implemented within the LINIF.

Naming conventions	
LinIf_	Services of LIN Interface
LinTp_	Services of LIN Transport Protocol
Lin_	Services of LIN Driver
LinTrcv_	Services of LIN Transceiver Driver
LinSm_	Services of LIN State Manager
PduR_	Services of PDU Router
Det_	Services of Development Error Tracer
BswM_	Services of Basic Software Mode Manager
EcuM_	Services of ECU State Manager

Table 2-1 Naming Conventions

2.2 Architecture Overview

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

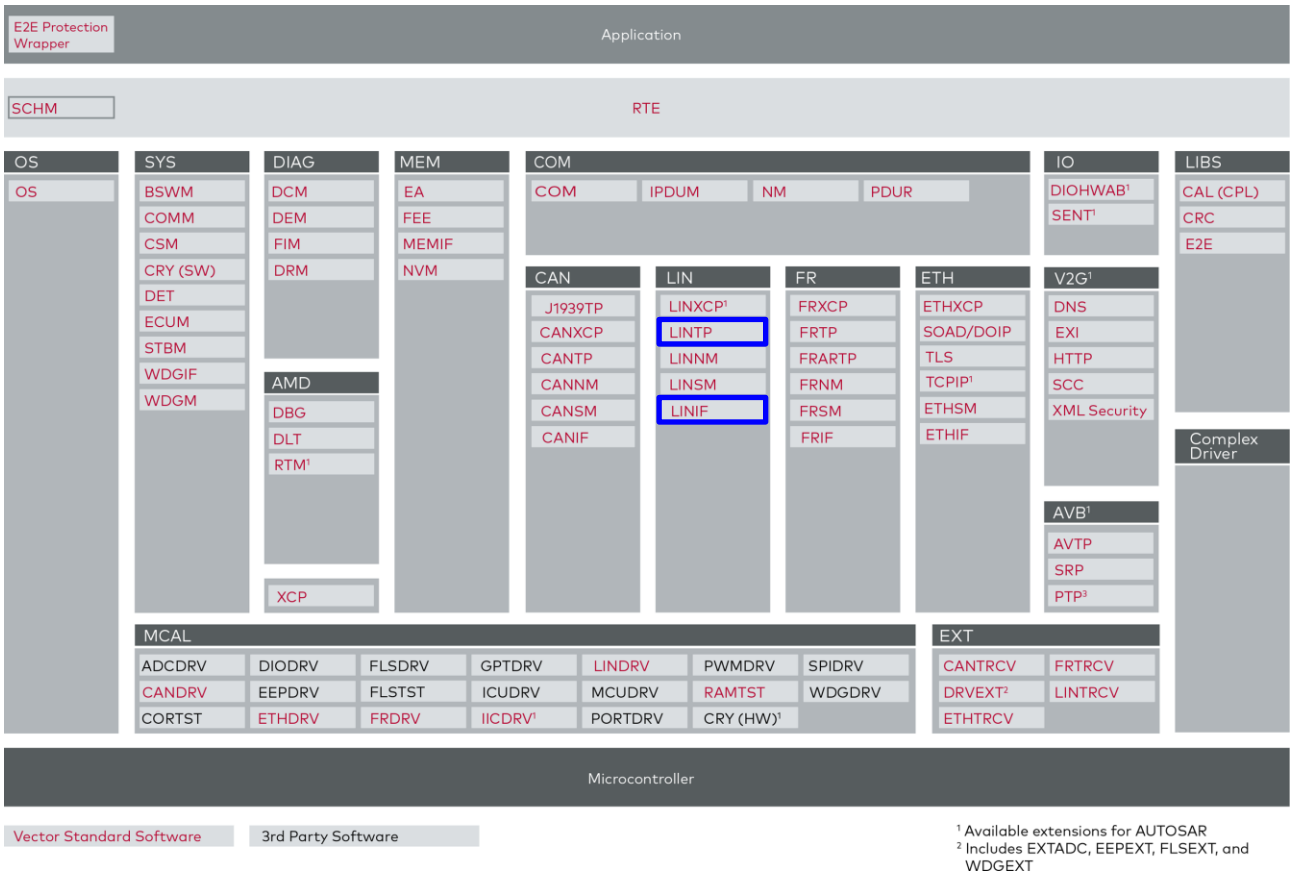

Figure 2-1 AUTOSAR 4.x Architecture Overview

The next figure shows the interfaces to adjacent modules of the LINIF. These interfaces are described in chapter 5.

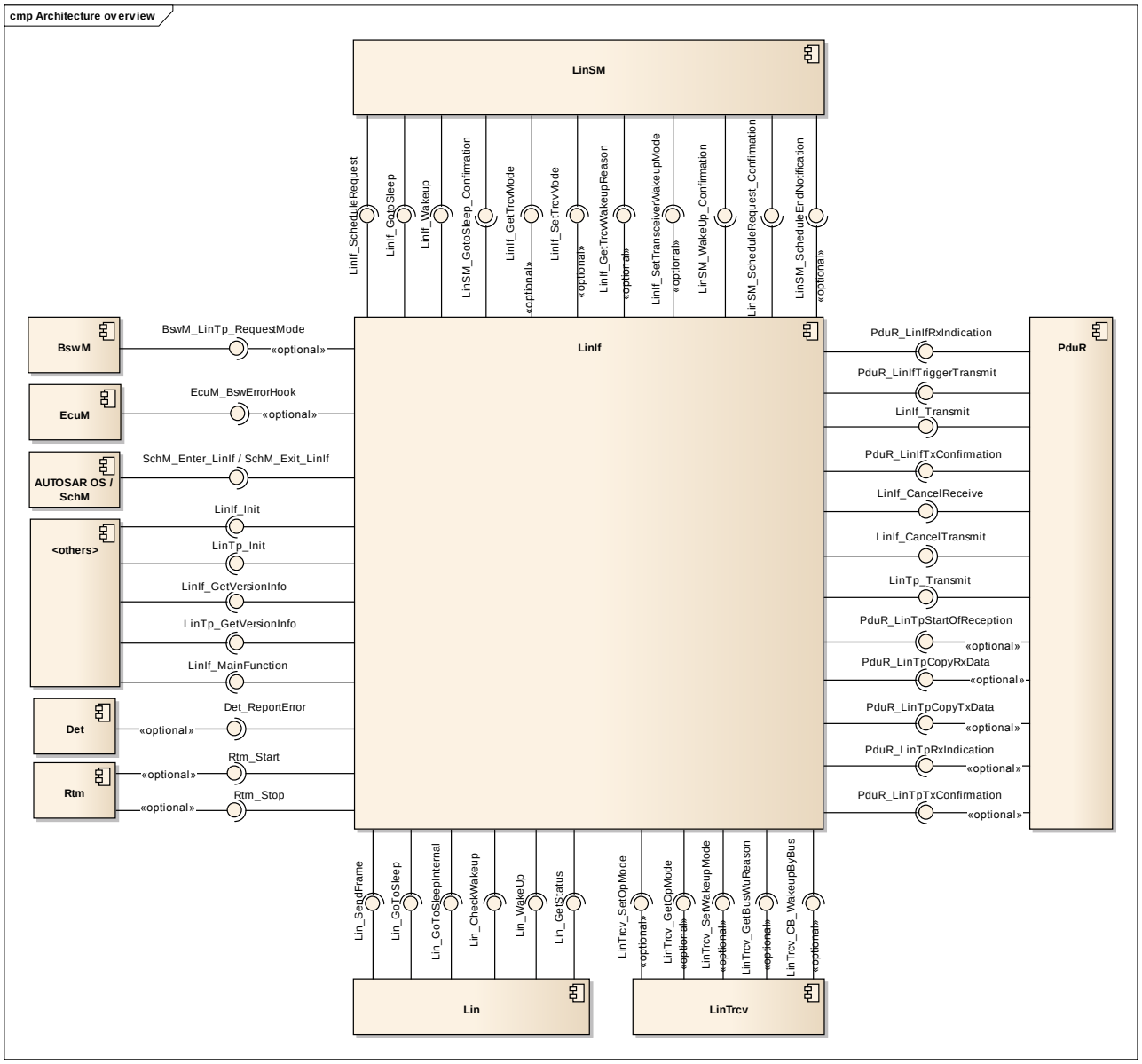

Figure 2-2 Interfaces to adjacent modules of the LINIF

Applications do not access the services of the BSW modules directly. They use the service ports provided by the BSW modules via the RTE. The LIN interface itself has no direct application access and therefore does not provide any service ports.

3. Functional Description

3.1 Features

The features listed in the following tables cover the complete functionality specified for the LINIF.

The AUTOSAR standard functionality is specified in [1], the corresponding features are listed in the tables

> Table 3-1 Supported AUTOSAR standard conform features

For information of not supported features see also chapter 7.

Vector Informatik provides further LINIF functionality beyond the AUTOSAR standard. The corresponding features are listed in the table

> Table 3-2 Features provided beyond the AUTOSAR standard

The following features specified in [1] are supported:

Supported AUTOSAR Standard Conform Features
LIN Interface initialization
(Multiple) LIN Driver handling
LIN Schedule table handling
LIN Interface main function handling
(Multiple) LIN Transceiver Driver support
Mode change: sleep, wakeup (with notification)
Interface to PDU Router
Interface to ECU Manager
Interface to LinSM
Development error detection and reporting to DET
Multi-channel support
Version reporting
Event-Triggert-Frames
Sporadic Frames
Transport Protocol Layer
Configuration variants
Node configuration
Schedule table support with run once and run continuous
Post-Build Loadable
External wakeup support according to AUTOSAR 4.1.x
MICROSAR Identity Manager using Post-Build Selectable

Table 3-1 Supported AUTOSAR standard conform features

The following features are provided beyond the AUTOSAR standard:

Features Provided Beyond The AUTOSAR Standard
Specification of internal and external wakeup delay before starting with scheduling
Handling of Response Pending Frames without PduR interaction
Optional handling of diagnostic functional requests
Automatic precompile optimizations
Explicit RAM initialization
Runtime Measurement Support
Load balancing between channels
Extended handling of diagnostic broadcast requests
Schedule End notification
Schedule Info API
LIN driver status callout

Table 3-2 Features provided beyond the AUTOSAR standard

3.2 Memory Initialization

AUTOSAR expects the startup code to automatically initialize the whole RAM content. However, no every startup code of embedded targets reinitializes all variables correctly. It is possible that the state of a variable may not be initialized as expected. To avoid this problem the MICROSAR LIN Interface provides an additional function to initialize all module variables.

Refer also to chapter 5.2.2 ‘LinIf_InitMemory’.

3.3 Initialization

After power-on, the LIN Interface has to be initialized. Therefore the LIN Interface provides two service functions.

The function `LinIf_InitMemory()` initializes all variables of the LIN Interface.

Caution
The LIN Interface assumes that some variables are initialized with zero at startup. If the embedded target does not initialize RAM after power on or reset (e.g. within the startup code), the function ‘`LinIf_InitMemory`‘ has to be called during start-up and before initialization of the LIN interface. Refer also to chapter 3.2 ‘Memory Initialization’.

The function `LinIf_Init()` initializes the channel independent and channel dependent states.

Note
LIN Driver and LIN Transceiver Driver are NOT initialized by the LIN Interface and shall be initialized before initialization of the LIN interface.

The channel initialization state can be configured using the feature ‘Startup State’. Each channel can be configured to NORMAL (operational) or SLEEP state. Please refer to chapter 4.6.1 for further details.

The function `LinTp_Init()` initializes global and channel dependent LINTP variables and sets the LINTP channel states to idle.

Caution
It is required to call `LinTp_Init()` after `LinIf_Init()`.

Note
In an AUTOSAR environment where the ECU Manager is used, the initialization is performed within the ECU Manager.

3.4 LIN Interface Operational Modes and States

The LIN interface has two global states:

- LINIF_UNINIT
- LINIF_INIT

By calling `LinIf_Init()`, the transition from LINIF_UNINIT state to LINIF_INIT state is performed.

Each channel has its own sub state machine which consists of the three following states:

- CHANNEL_UNINIT
- CHANNEL_OPERATIONAL
- CHANNEL_SLEEP

After power-on, each channel is in state CHANNEL_UNINIT. Depending on the configured startup state, the transition to CHANNEL_SLEEP or CHANNEL_OPERATIONAL is performed during channel initialization according to Figure 3-1.

The CHANNEL_OPERATIONAL state can be directly entered after initialization, depending on the startup state configuration. In case the channel is in CHANNEL_SLEEP state, a transition to CHANNEL_OPERATIONAL can be requested by calling LinIf_Wakeup(). See 3.7 ‘Wake up Handling’ for further details.

Note
The transition from CHANNEL_SLEEP to CHANNEL_OPERATIONAL is not performed in `LinIf_CheckWakeup()` as defined in [1] but in Lin_Wakeup.

The CHANNEL_SLEEP state can be directly entered after initialization, depending on the startup state configuration. In case the channel is in CHANNEL_OPERATIONAL state, a transition to CHANNEL_SLEEP can be requested by calling LinIf_GotoSleep (). See 3.8 ‘Sleep Handling’ for further details.

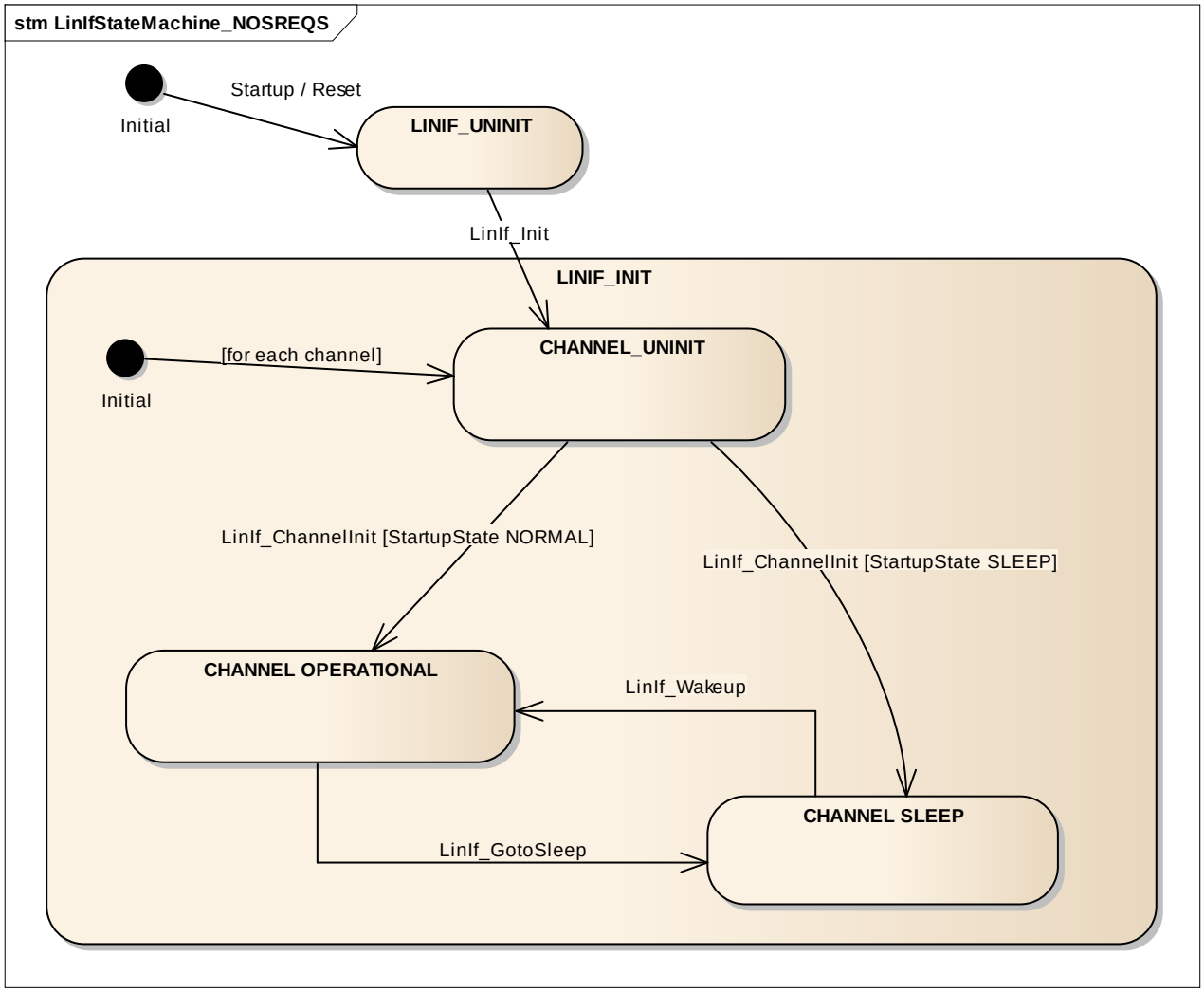

Figure 3-1 LinIf State machine

3.5 LIN Transport Protocol States

The LinTp provides its own global and channel-dependent state machine (Figure 3-2).

The LIN transport protocol has two global states:

- LINTP_UNINIT

> LINTP_INIT

By calling `LinTp_Init()`, the transition from LINTP_UNINIT state to LINTP_INIT state is performed and each channel state is set to idle.

Each channel has its own sub state machine which consists of the three following states:

> LINTP_CHANNEL_IDLE

> LINTP_CHANNEL_BUSY

In case of MRF transmission, the channel state is set to LINTP_CHANNEL_BUSY (TX) busy if the transmission data was successfully requested and set back to LINTP_CHANNEL_IDLE if the complete request was transmitted or an error occurred.

If a valid response frame was received for a scheduled SRF header, a new reception is started and the state transition to LINTP_CHANNEL_BUSY (RX) is performed. If all expected data is received or an error occurred, the state is set to LINTP_CHANNEL_IDLE.

Note that a new TP transmission request will abort an ongoing TP reception or ongoing TP transmission and accept the new TP transmission request.

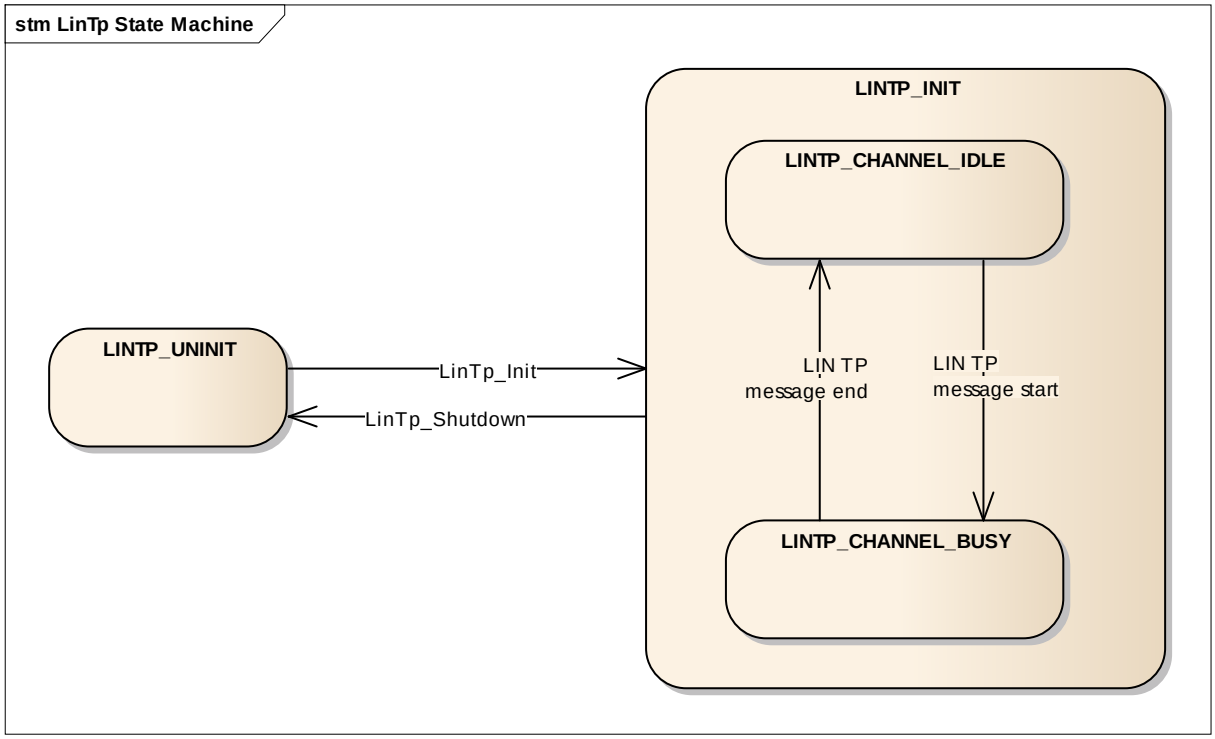

Figure 3-2 LinTp State machine

3.6 Main Functions

The LIN interface provides the main function `LinIf_MainFunction()` which performs the schedule and frame handling for all channels including transport protocol handling.

Each channel has its own channel time base defined by parameter ‘LinIfClusterTimeBase’ which is by default the time base of the initial underlying database (LDF).

The global cycle time of the main function is configured by the parameter ‘LinIfTimebase’ and must be a common integer divisor of all ‘LinIfClusterTimeBase’ values respectively each channel time base must an integer multiple of the main cycle time.

For best efficiency, it is recommended that the LinIfTimebase value is the greatest common divisor of all channel time bases.

The integrator must ensure that `LinIf_MainFunction()` is called with the configured global cycle time and minimal jitter to guarantee correct frame handling. The LinIf handles internally the different time bases of the channels.

3.7 Wake up handling

Two different types of wakeup frame handling are distinguished: Internal or External.

Internal wakeup frame handling (no LIN wakeup frame was previously detected on the current channel) depends on the current channel state:

- CHANNEL_SLEEP: Wakeup frame handling is only applicable in this state. A wakeup frame can be transmitted by calling the function `LinIf_Wakeup()`.This will request the LIN driver to transmit a wakeup frame. The successful transmission is confirmed to the upper layer by calling `<User>_WakeupConfirmation()`.

  After the wakeup is transmitted, the duration until the actual scheduling is started can be configured by the option ‘LinIf_WakeupDelay’ for each channel separately.
- CHANNEL_OPERATIONAL: If `LinIf_Wakeup()` is called in this state, nothing will be done and a successful wakeup is indicated to upper layer.

An external wakeup on the LIN bus is detected either by the LIN driver or the LIN transceiver (if available). Following sequence is performed in this case:

- LIN driver or LIN transceiver informs EcuM about an external wakeup event.
- EcuM notifies LINIF by calling `LinIf_CheckWakeup()`.
- Depending on the given wakeup source parameter in `LinIf_CheckWakeup()`, the corresponding <*>_CheckWakeup function of the underlying LIN transceiver or LIN driver is called.
- If the wakeup source was validated in the <*>_CheckWakeup function, `LinIf_WakeupConfirmation()` is called as well as the EcuM is notified.
- After the wakeup was validated through the AUTOSAR stack, `LinIf_Wakeup()` is called. In this case, no further wakeup frame is transmitted but a successful wakeup is indicated to upper layer.

3.8 Sleep handling

Sleep mode frame handling is only applicable in state CHANNEL_OPERATIONAL. A sleep mode frame can be transmitted by calling the function `LinIf_GoToSleep()`.The currently active schedule table slot is normally processed and finished. In the successive slot, a sleep frame is scheduled (instead of the actual header specified in the schedule

table). After the transmission of the sleep frame, the state CHANNEL_SLEEP is reached and the upper layer is informed by calling `<User>_GotoSleepConfirmation()`.

3.9 Schedule Table Handling

LINIF is responsible for schedule table handling. The NULL schedule table (containing no frames) is always available for each channel and is set as active schedule table during initialization. It has always the schedule table index zero.

Each schedule has one of the two run types, RUN_CONTINUOUS or RUN_ONCE. RUN_CONTINUOUS implies that the active schedule table is processed until another table is requested. If the end of the schedule table is reached, it is automatically restarted at the beginning. A RUN_ONCE schedule table is processed once from beginning to end. If the end is reached, the schedule table is changed back to the last RUN_CONTINUOUS schedule table or NULL schedule table in case no RUN_CONTINUOUS table has ever been requested yet.

During runtime, the service function `LinIf_ScheduleRequest()` can be used by upper layer to set a new schedule table. The request is stored and the new schedule table is activated depending on the run type of the currently active schedule table:

a) If the current schedule is of type RUN_CONTINUOUS, the schedule table is switched after the current frame slot is finished.

b) If the current schedule is of type RUN_ONCE, the complete schedule table is processed once and the switch is done when the last frame slot of the RUN_ONCE schedule is finished. If there several schedule requests during the processing of a RUN_ONCE schedule, the last requested table is taken and former requests are overruled.

c) Requesting the NULL schedule table is special in that way that it is always set active at the next frame slot, even if the current schedule is of type RUN_ONCE.

The exact point in time when a requested schedule table is activated depends on the parameter ‘Schedule Change Next TimeBase’:

a) If disabled, the actual schedule table switch is performed at the end of the current schedule table slot.

b) If enabled, the actual schedule table switch is performed when the message transmission / reception is finished within the current schedule table slot. In general. this is before the end of the actual slot.

In both cases, the run type of the current schedule table is also considered.

If a collision for an EVT frame is detected, the LINIF has to switch to the corresponding collision resolving schedule table. If the collision occurs in a RUN_ONCE schedule, it is not interleaved but the actual switch is postponed until the RUN_ONCE schedule table is run-through. This implies that currently only one EVT frame is allowed per RUN_ONCE schedule.
If there is already a regular schedule table change pending when the collision is detected, the switch to the collision resolving table is discarded. Instead the requested schedule table is going to be set to active after the EVT slot.

Because a collision resolving schedule table is not requested from outside, there is no confirmation reported for the collision resolving schedule switch by the function `<User>_ScheduleRequestConfirmation()`.

By enabling the feature ‘Schedule End Notification’, the complete run-through of a schedule table is notified to upper layer. If the last slot of the current schedule table is going to be started, the callback LinSM_ScheduleEndNotification() is triggered with the handle of the current schedule table. Note that this handling is performed for all schedules tables except the NULL schedule, but including RUN_ONCE and collision resolving schedule tables. This notification can be used for rule definitions in the BswM to allow a switch to another schedule table right at the end of a schedule.

FAQ
It is not possible to configure a start-up schedule table inside the LIN Interface module. This functionality can be implemented by a rule in the BswM module, see [5].

3.9.1 Schedule Table Handling after Wakeup

According to [6], each slave node shall be ready to listen to bus commands within 100ms, measured from the end of the wake up frame. This requirement provides a LIN node enough time for initialization after wakeup frame reception before the normal bus communication starts.

Two additional parameters are added to MICROSAR LINIF to support a configurable wakeup delay for each channel separately:

- Parameter ‘LinIf_WakeupDelay’ configures the duration after an internal wakeup request
- Parameter ‘LinIf_WakeupDelayExternal’ configures the duration after an external wakeup frame was detected on the LIN bus

In both cases, after the transition to CHANNEL_OPERATIONAL state, the actual scheduling is not started until the configured wakeup delay has elapsed.

These delays can be disabled (not recommended!) by setting the particular parameter to a value smaller than the LinIfTimebase parameter.

3.10 LIN Transport Protocol

The LIN TP is used to transport Diagnostic services and responses. No Diagnostic sessions are made in parallel. All service requests are always proceeded in sequence, with exception of functional requests (refer to 3.10.1 for further details).

For LinTp connections the two dedicated frames MasterReq and SlaveResp are used exclusively. Each connection is defined by the parameters LIN channel and a node address for diagnostics (NAD).

For transmission of a diagnostic request, a TxNsdu has to be configured on the channel. All existing TxNsdus on a channel share the MasterReq frame for transmission.

Analogous, for reception of a diagnostic response an RxNsdu has to be configured and all RxNSdus on a channel share the SlaveResp frame for reception.

Only one (physical) LinTp connection can be active at a certain time. An ongoing LinTp connection is aborted inside a `LinTp_Transmit`() call and the new diagnostic request is **accepted.**

A diagnostic reception is by default only accepted if the NAD matches the NAD of the request. Disable ‘LinTpDropNotRequestedNad’ to accept all diagnostic receptions independent of the NAD.

3.10.1 Functional Requests

A functional request (NAD = 0x7E) may interleave the physical diagnostic communication at any time. The functional request is transmitted always first when pending and no answer is expected. The handling of functional requests can be disabled if not required by parameter ‘LinTpFunctionalRequestsSupported’ for optimization reasons.

3.10.2 Response Pending Frames

Extended timeout observation based on parameters ‘LinTpP2Timing’ and ‘LinTpP2Max’ (refer also to chapter 3.12) check the timing of the slave node after a diagnostic request was transmitted successfully. The slave node may reply to a request with a response pending (RP) command in case the P2 timeout is not sufficient. The layout of a response pending frame is as follows:

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
<NAD>	0x03	0x7F	<SID>	0x78	0xFF	0xFF	0xFF

Table 3-3 Layout of response pending frames

After reception of a RP frame, the diagnostic timeout observation is restarted using the P2Max timeout instead of the P2 timeout value. A slave node may respond with one or several RP frames. If the number of received RP frames exceeds ‘LinTpMaxNumberOfRespPendingFrames’, the connection is aborted by the LinTp.

The option ‘Forward Response Pending Frames’ configures if RP frames are forwarded to PduR (recommended, especially for routing use cases) or handled solely by LinTp.

3.10.3 Broadcast Requests

A broadcast request (NAD = 0x7F) is received by every slave node. If subsequently an SRF header is scheduled, more than one slave node might respond and a collision would occur. So the master node may choose not to ask for the response from the slave node.

If parameter ‘LinTpBroadcastRequestHandling’ is enabled, an applicative schedule table is requested and no slave response is expected after a valid transmission of a broadcast request. If disabled, a broadcast requests is handled like any other diagnostic request, so a schedule table switch to a diagnostic response schedule table is requested and a P2 timeout is started.

Note that in order to receive a response for a broadcast request (e.g. if only one slave responds), beside ‘LinTpBroadcastRequestHandling’ also ‘LinTpDropNotRequestedNad’ has to be disabled.

3.11 Diagnostic Schedule Table Change

The purpose of the parameter ‘LinTpScheduleChangeDiag’ is to enable a notification to the upper layer according to the diagnostic communication demand.

If this feature is enabled, LinTp will call `BswM_LinTp_RequestMode()` in order to trigger a schedule change request depending on the required diagnostic mode. There are three cases distinguished by the argument passed to BswM:

> LINTP_DIAG_REQUEST: A schedule table with a MRF frame is requested

When a diagnostic request is accepted by LinTp_Transmit(), BswM_LinTp_RequestMode() is called with parameter `LINTP_DIAG_REQUEST` so that MRFs are scheduled to transmit the TP data.

> LINTP_DIAG_RESPONSE: A schedule table with a SRF frame is requested

When the request is transmitted and the response may be pending, a new schedule change request is done with parameter `LINTP_DIAG_REQUEST`.

> LINTP_APPLICATIVE_SCHEDULE: An applicative schedule is requested

At the end of reception or on occurrence of any kind of error, the parameter `LINTP_APPLICATIVE_SCHEDULE` is used in order to allow the upper layer to switch back to the latest normal schedule table.

If the BswM does not follow the requested schedule no diagnostic communication will take place. This can force time out conditions. To change the schedule table the LinSM API `LinSM_ScheduleRequest()` should be used.

In the following figure the diagnostic communication and reporting of events to the upper layers is depicted. To have a constant and small jitter in the scheduling any notification to upper layers like the provide port of LinTp is done after transmitting the next frame of the schedule. This will mean a delay of one frame before a schedule table change can take effect. As a result of this optimization one slot between a diagnostic request and its response can be empty. Also a slave response header may be sent without response at the end of a diagnostic response before the normal schedule will be preceded. Those events do not have any side effect to the communication except timing. If there is at least two T_{base} cycles after a frame transmission and the start of the next frame (schedule table delay) no additional delays will take place.

FAQ
The schedule table change is configured by a rule in the BswM module, see [5].

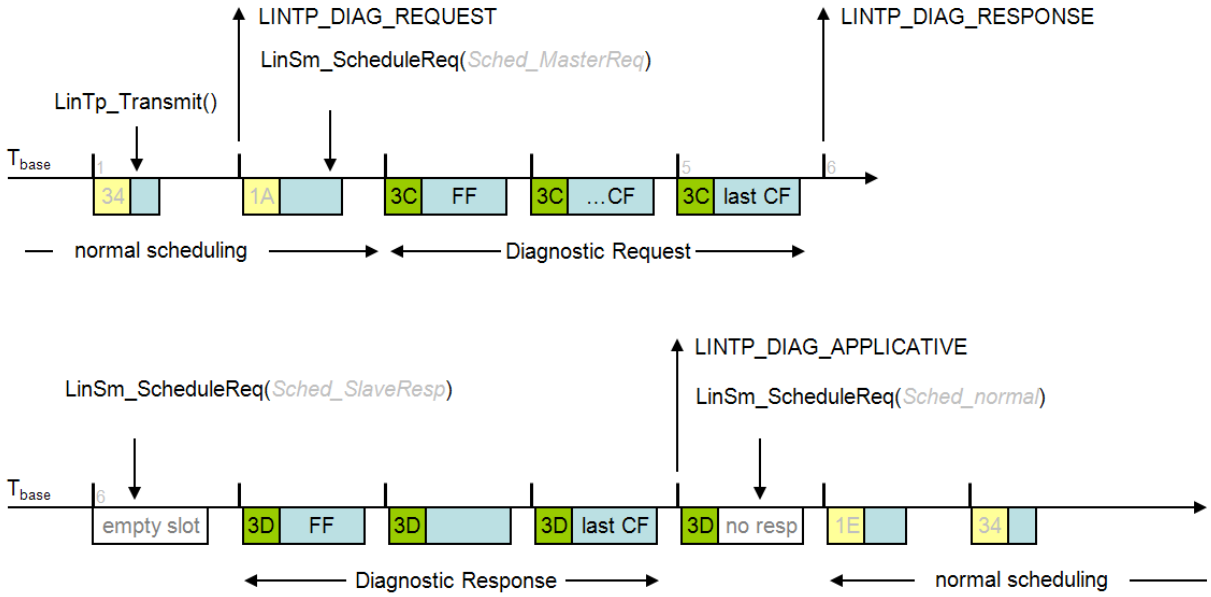

Figure 3-3 Diagnostic communication and schedule table changes

3.12 Transport Protocol Timeout Handling

Each LinTp connection is supervised by a timeout handler. This is required to prevent deadlock situations, e.g. when the diagnostic schedule table change (see 3.11) is not correctly implemented and no Master Request frame (Tx) / no Slave Response Frame (Rx) is scheduled after a transmission request / awaited response answer. Another scenario could be a request to a not-assembled slave node – without timeout supervision, the LinTp connection would get stuck waiting forever for a response from the slave.

The LinTp provides several timeout parameters for the different sub-states of a LinTp connection. They can be grouped into one of the two categories:

- Transport Layer timing constraints
- Diagnostic timing constraints

Transport layer timing parameters are configured per Tx/Rx connection and are defined in Table 3-4:

Parameter	Direction	Timer start	Timer end
LinTpNas	Tx	Transmission request via LinTp_Transmit() accepted.	Diagnostic frame (SF or FF) has been transmitted.
LinTpNcs	Tx	Diagnostic frame in the same message has been transmitted.	Next CF is requested for transmission.
LinTpNcr	Rx	Diagnostic frame (FF or CF) has been received.	Next diagnostic frame in the same message (CF) has been received.

Table 3-4 Transport Layer timing constraints

Diagnostic timing parameters are globally configured for the LinTp and are defined in Table 3-5:

Parameter	Timer start	Timer end
LinTpP2Timing	Last frame of a diagnostic request (SF or CF) has been transmitted.	Diagnostic response frame (SF or FF) has been received.
LinTpP2Max	A response pending frame has been received.	Diagnostic response frame (CF) has been received.

Table 3-5 Diagnostic timing constraints

In case a timeout condition occurs, the currently active LinTp connection is aborted, PduR is accordingly notified and a schedule table change back to applicative schedule is triggered.

Following figure visualizes the different phases with active timeout observation:

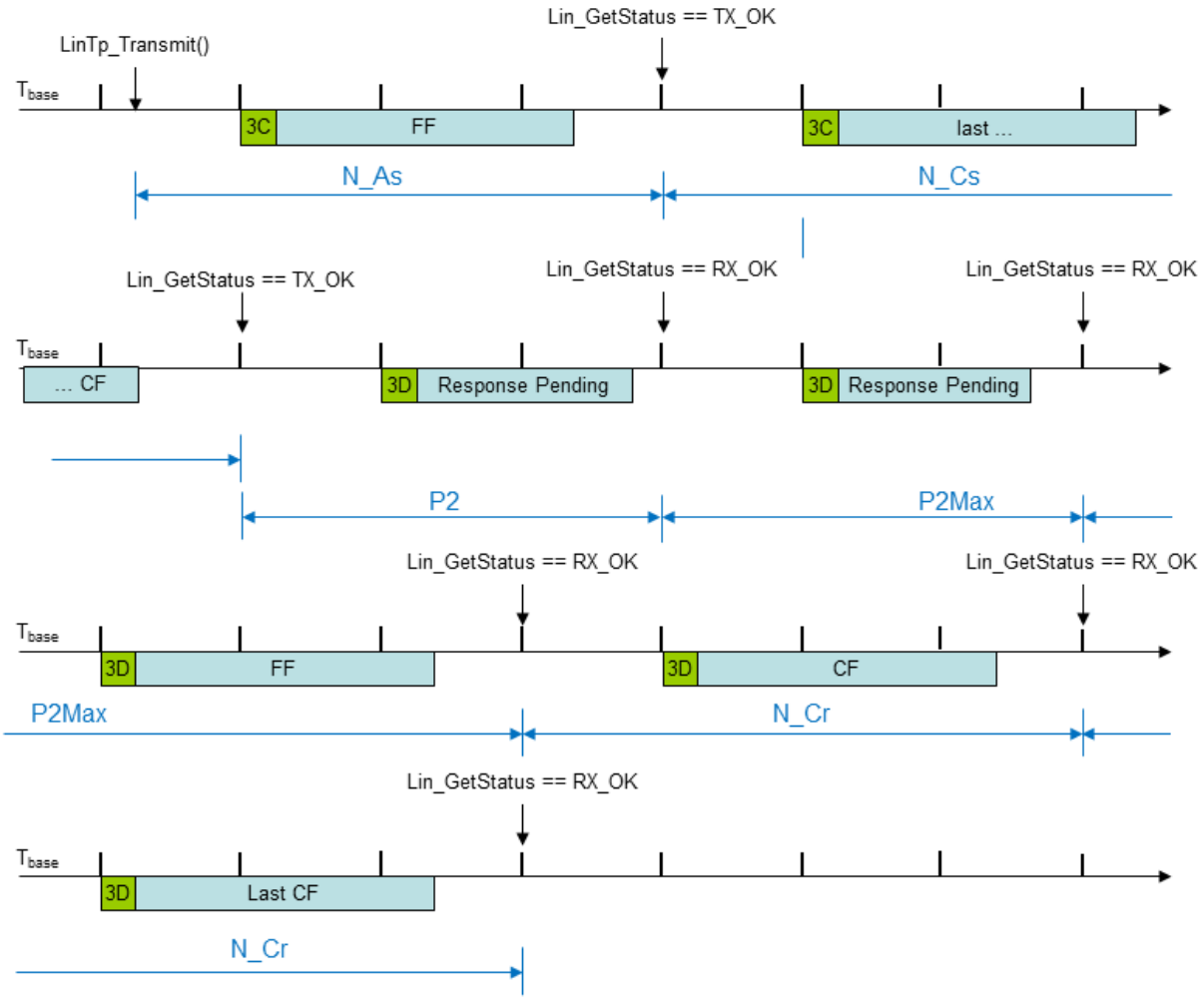

Figure 3-4 LinTp Timeouts

3.13 Diagnostic Communication

Diagnostic communication in LINIF can be distinguished between:

> LIN Node Configuration Services

> Diagnostics over LIN Transport Protocol

LIN Node Configuration Services are supported via fixed schedule table commands. Each schedule table command is defined in the configuration by a particular LIN frame type, e.g. ASSIGN_NAD, FREEFORMAT. The eight bytes of each request (Node configuration commands are always single frames) are provided fixed in the configuration, thus the transmission is solely handled by LINIF without upper layer interaction. Consequently, a response to a LIN Node Configuration command is also not forwarded to upper layer.

Note
Autosar does not specify an interface to the response of a node configuration command as there is either a positive or no response at all. Anyhow, the response data does not contain useful information; therefore the response is completely handled inside LINIF.

FAQ
In case the application requires "dynamic" node configuration requests or access to the slave response, the node configuration communication has to be manually handled by application over the transport protocol using a Complex Device Driver. Fixed schedule table commands are then not applicable.

FAQ
Master Request and Slave Response Frames are only accessible with LIN Transport Protocol. There is no direct interface to access MRF / SRF using solely LINIF.

LIN Transport Protocol can be used for application-specific diagnostics. Therefore the access to the LIN diagnostics frames is realized using a Complex Device Driver. This is accomplished by configuring a connection between CDD, PDUR and LINTP. Here an overview of necessary configuration steps required for each addressed slave node:

> Add a Tx and a Rx NSDU container in LinTp

> Add global PDUs in EcuC module for both NSDUs

> Add a CDD module to the configuration (if none exists) and configure the transport protocol contribution appropriately (for details, please refer to see [13])

> Add routing paths for Tx and Rx direction to PduR to connect CDD and LinTp

The application use the services provided by the CDD to request a diagnostic transmission to a slave node and to get notified about the received payload bytes of the diagnostic slave response.

3.13.1 Diagnostic Communication Dispatching

Because the MRF and SRF are shared between gateway / application diagnostics and NCC, the LinTp must dispatch the access of those diagnostic frames to the correct user.

The dispatcher uses a priority based assignment where Node Configuration Services have a higher priority than diagnostics over transport protocol. An ongoing LinTp connection is aborted if a schedule table slot with a LIN node configuration service is activated.

3.14 Error Handling

3.14.1 Development Error Reporting

By default, development errors are reported to the DET using the service `Det_ReportError()` as specified in [2], if development error reporting is enabled (i.e. pre-compile parameter `LINIF_DEV_ERROR_DETECT==STD_ON`).

If another module is used for development error reporting, the function prototype for reporting the error can be configured by the integrator, but must have the same signature as the service `Det_ReportError()`.

The reported LINIF ID is 62.

The reported service IDs identify the services which are described in 5.2. The following table presents the service IDs and the related services:

Service ID	Service
0x01	`LinIf_Init`
0x03	`LinIf_GetVersionInfo`
0x04	`LinIf_Transmit`
0x05	`LinIf_ScheduleRequest`
0x06	`LinIf_GotoSleep`
0x07	`LinIf_Wakeup`
0x08	`LinIf_SetTrcvMode`
0x09	`LinIf_GetTrcvMode`
0x0A	`LinIf_GetTrcvWakeupReason`
0x0B	`LinIf_SetTrcvWakeupMode`
0x0C	`LinIf_CancelTransmit`
0x0D	`LinIf_GetScheduleInfo`
0x40	`LinTp_Init`
0x41	`LinTp_Transmit`
0x42	`LinTp_GetVersionInfo`
0x43	`LinTp_Shutdown`
0x44	`LinTp_ChangeParameter`
0x46	`LinTp_CancelTransmit`
0x47	`LinTp_CancelReceive`
0x60	`LinIf_CheckWakeup`
0x61	`LinIf_WakeupConfirmation`
0x80	`LinIf_MainFunction`

Table 3-6 Service IDs with mapped services

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

Error Code		Description
0x00	LINIF_E_UNINIT	API called without initialization of LIN Interface.

Error Code		Description
0x10	LINIF_E_ALREADY_INITIALIZED	Initialization API is used when already initialized.
0x20	LINIF_E_NONEXISTENT_CHANNEL	Referenced channel does not exist (identification is out of range).
0x30	LINIF_E_PARAMETER	API service called with wrong parameter.
0x40	LINIF_E_PARAMETER_POINTER	API service called with invalid pointer.
0x51	LINIF_E_SCHEDULE_REQUEST_ERROR	Schedule request made in channel sleep state.
0x52	LINIF_E_TRCV_INV_CHANNEL	Referenced channel does not exist (identification is out of range) or no transceiver exist on referenced channel.
0x53	LINIF_E_TRCV_INV_MODE	Given transceiver mode does not exist.
0x54	LINIF_E_TRCV_NOT_NORMAL	Transceiver API call during unexpected transceiver state.
0x55	LINIF_E_PARAM_WAKEUPSOURCE	LinIf_CheckWakeup called with invalid wakeup source parameter value.
0x60	LINIF_E_RESPONSE	An error occurred within the response of a received or transmitted message.
0x61	LINIF_E_NC_NO_RESPONSE	No diagnostic answered has been received from the LIN Slave after transmission of a node configuration command.
0x70	LINIF_E_SCHEDULE_INCONSISTENT_ERROR	Frame length exceeds schedule slot length. Has been introduced by Vector Informatik GmbH.
0x71	LINIF_E_CONFIG	Invalid configuration, because configuration structure not valid. Has been introduced by Vector Informatik GmbH.
0x72	LINIF_E_TRIGGERTRANSMIT_NO_DATA	<User>TriggerTransmit was unable to provide transmission data. No frame will be send; unconditional frame slot will remain empty.

Table 3-7 Errors reported to DET

3.14.1.1 Parameter Checking

AUTOSAR requires that API functions check the validity of their parameters. The checks in Table 3-8 are internal parameter checks of the API functions. These checks are for development error reporting and can be en-/disabled by means of en-/disabling the DET reporting via the parameter `LINIF_DEV_ERROR_DETECT`.

The following table shows which parameter checks are performed on which services:

Service \ Check	ConfigPtr	versioninfo/scheduleinfo	Channel	LinTxPduId	PduInfoPtr	Schedule	LinTpTxSduId	LinTpTxInfoPtr	TransceiverMode(Ptr)	TrcvWakeupMode/Reason	WakeupSource
LinIf_Init()	■										
LinIf_GetVersionInfo()		■									
LinIf_Transmit()				■	■						
LinIf_ScheduleRequest()			■			■					
LinIf_GotoSleep()			■								
LinIf_Wakeup()			■								
LinIf_SetTrcvMode()			■						■		
LinIf_GetTrcvMode()			■						■		
LinIf_GetTrcvWakeupReason()			■							■	
LinIf_SetTrcvWakeupMode()			■							■	
LinIf_CancelTransmit				■							
LinIf_GetScheduleInfo		■	■								
LinTp_Init()	■										
LinTp_Transmit()							■	■			
LinTp_GetVersionInfo()		■									
LinTp_Shutdown()											
LinTp_ChangeParameter()							■				
LInTp_CancelReceive()							■				
LinTp_CancelTransmit()							■				
LinIf_CheckWakeup()											■
LinIf_WakeupConfirmatiom											■
LinIf_MainFunction()											

Table 3-8 Development Error Reporting: Assignment of checks to services

3.14.2 Production Code Error Reporting

The LIN interface does not report any production errors to DEM.

3.14.3 LIN Communication Error Detection

During development, short-term communication problems are detected by evaluating the status value of the LIN driver and are reported to the DET. They are caused by erroneous, missing or incomplete response parts of LIN frames. These error events help to identify

integration faults like e.g. missing port configuration, incorrect call cycle of LINIF main function or jitter, wrongly configured baudrate or misconfigured interrupt table.

During service and production however, the fundamental idea for detecting problems on physical layer is that the error identification is completely handled on communication layer (AUTOSAR COM module and SWCs). Table 3-9 summarizes various error types and how they can be detected by Com/SWC:

Error Type	Detection by Com/SWC
Electrical problem, e.g. ▶ Short-circuit to ground ▶ Short-circuit to U_{batt}	Tx timeout
No bus communication, e.g. ▶ interruption of the LIN bus line	Rx timeout of all slaves
No communication to a specific slave node, e.g. ▶ slave node defect	Rx timeout of all signals of a slave node
Local disturbance detected by master node, e.g. ▶ one or more bits are disturbed ▶ wrong checksum	Rx timeout of some but not all signals of a slave node
Local disturbance detected by slave node, e.g. ▶ one or more bits are disturbed ▶ wrong checksum	Response error bit set (signal access and evaluation on SWC level)

Table 3-9 Communication error types and detection

For further information about COM timeout monitoring, please refer to [12].

Expert Knowledge
In order to get the LIN driver status on frame level, the callout function `Appl_LinIfGetLinStatus()` could additionally be used. See 5.6.1 for further details.

3.15 Load Balancing by configurable call cycle offsets

The LINIF supports load balancing to evenly distribute the total runtime load of the particular channels.

`LinIf_MainFunction` is the only task function of the LINIF which handles all LIN channels, therefore its runtime increase with the number of channels. Additionally, the

more LIN channels are used with the same tick time, the more likely will bursts of runtime occur.

The reason is that in some ticks only very less processing has be done – this is e.g. the case if a LIN frame is currently transmitted on the bus. In other main function calls however, the required handling multiplies (e.g. handling of the last frame, running the schedule manager, reporting to upper layers etc.). This is especially the case when a new slot on several channels is due caused by equal cycle times. Figure 3-5 depicts this issue:

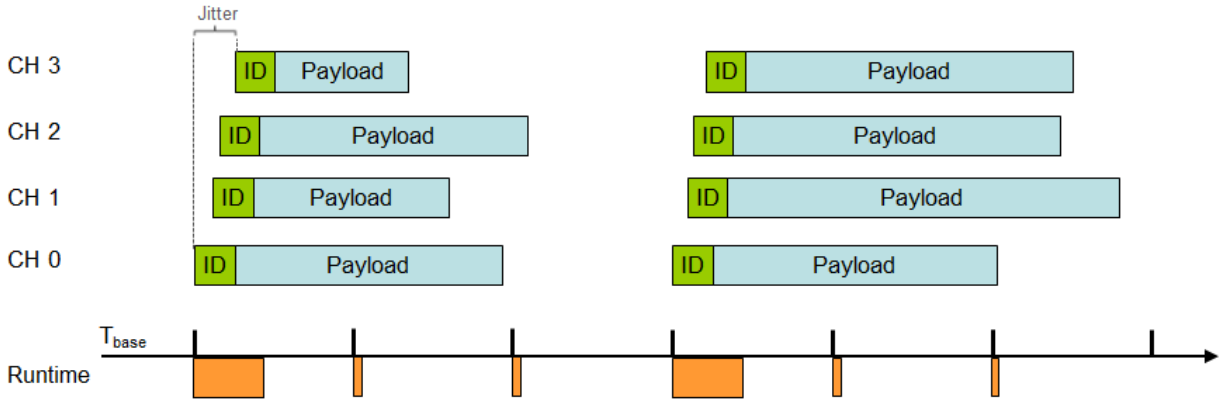

Figure 3-5 Runtime bursts on multi-channel systems with same cycle time

Additionally to the significant variance of the runtime, the jitter between the beginning of the schedule slot and the actual transmission start of the frame also extends.

Load balancing reduces these effects by using a channel-dependent delay offset. The number of channels to be handled in the same tick is minimized. The smaller the global call cycle time of the `LinIf_MainFunction` is chosen, the better the runtime distribution is harmonized. Also the worst execution time of the main function can be lowered.

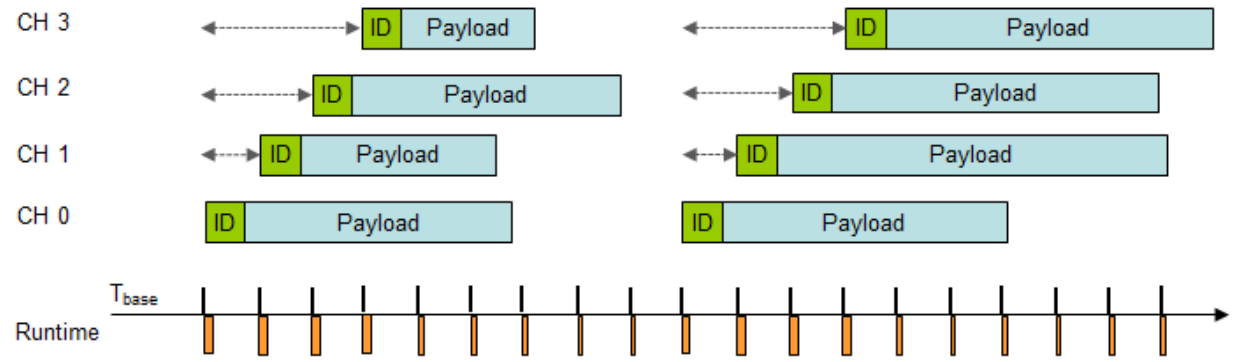

Figure 3-6 Load Balancing visualization

Caution
Due to the asynchronous execution of the channel handling, Load Balancing cannot be combined with synchronous data provision as described in 4.6.3.

To enable the load balancing feature, specify on at least one channel a load balancing offset greater than 0 (LinIfLoadBalancingOffset parameter). The load balancing offset of a

channel is the duration in milliseconds until the actual handling of the channel is started after initialization of LINIF. It is recommended that this value is smaller than the channel/cluster time base. In case of mismatches with the global or channel time base, a live validation error is reported. The feature is disabled if the offset on all channels is configured to 0 (default case).

Expert Knowledge
If a channel transits from SLEEP to OPERATIONAL mode while the load balancing delay has not yet expired (after initialization of LINIF), the wakeup delay (refer to 3.9.1) increases by the remaining load balancing delay. Another effect is that an internal wakeup request cannot be finished while to initial load balancing delay has not expired, so the whole wakeup process is delayed.

Note
It is highly **recommended** to choose **values** smaller than the cluster time base of the channel for the load balancing offset.

3.16 SAE J2602 Frame Tolerance

The LinIf supports a frame timeout definition according to SAE J2602 where for each LIN node a dedicated maximum frame tolerance can be specified. If this feature is enabled by the general SAE J2602 frame tolerance support switch, then following additional configuration parameters are required:

- The maximum header tolerance of the master per channel in bits
- The maximum response tolerance of the master per channel in percent
- The maximum response tolerance of each Rx frame. As the slave nodes are not modelled in the LinIf configuration, the response tolerance of a slave node is configured for each of its transmitted frames.

The tolerance values are evaluated to validate the schedule table slot delays and to calculate the optimized point in time to poll the LIN driver for the status of the current schedule table slot.

Expert Knowledge
The response tolerance of a slave response frame schedule slot is equal to the maximum configured response tolerance value of all receive frames.

The response tolerance of an event-triggered frame schedule slot is equal to the maximum response tolerance value of its assigned unconditional frames.

4. Integration

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

4.1 Scope of Delivery

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

4.1.1 Static Files

File Name	Description
LinIf.h	Header containing the interface of the LIN Interface.
LinIf.c	C code containing the functionality of the LIN Interface. This file is either delivered as source code or as library. If delivered as library the name is LinIf.* with the compiler specific library extension.
LinIf_Cbk.h	Header containing the function prototypes of the callback functions.
LinIf_Types.h	Header containing type definitions of the LIN Interface.
LinTp_Types.h	Header containing type definitions of the LIN Transport Layer. This file is optional and only required if the LIN Transport Layer is used.

Table 4-1 Static files

Do not edit manually
The static files in Table 4-1 and generated files in Table 4-2 / Table 4-3 must not be edited by the user!

4.1.2 Dynamic Files

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

File Name	Description
LinIf_Cfg.h	Generated header adapting the LIN Interface to project requirements.
LinIf_Lcfg.c	Generated C code containing tables with link time variables (RAM/ROM).
LinIf_PBcfg.c	Generated C code containing tables with post build variables (ROM).
LinIf_Lin.h	Generated header adapting to the LIN Driver include header(s).
LinIf_LinTrcv.h	Generated header adapting to the LIN Transceiver Driver include header(s).
LinIf_GeneralTypes.h	Generated header containing type definitions of the LIN interface used by other modules. Included by general static header Lin_GeneralTypes.h.

Table 4-2 Generated files for LIN Interface

In case of active TP the following dynamic files are also generated by the configuration tool GENy.

File Name	Description
LinTp_Cfg.h	Generated header adapting the LIN TP to project requirements.
LinTp_Lcfg.c	Generated C code containing tables with link time variables (RAM/ROM) of the LIN TP.
LinTp_PBcfg.c	Generated C code containing tables with post build variables (ROM) of the LIN TP.

Table 4-3 Generated files for LIN Transport Protocol

Having separate generated files for LinIf and LinTp allows mapping the configuration data to different memory locations.

4.2 Include Structure

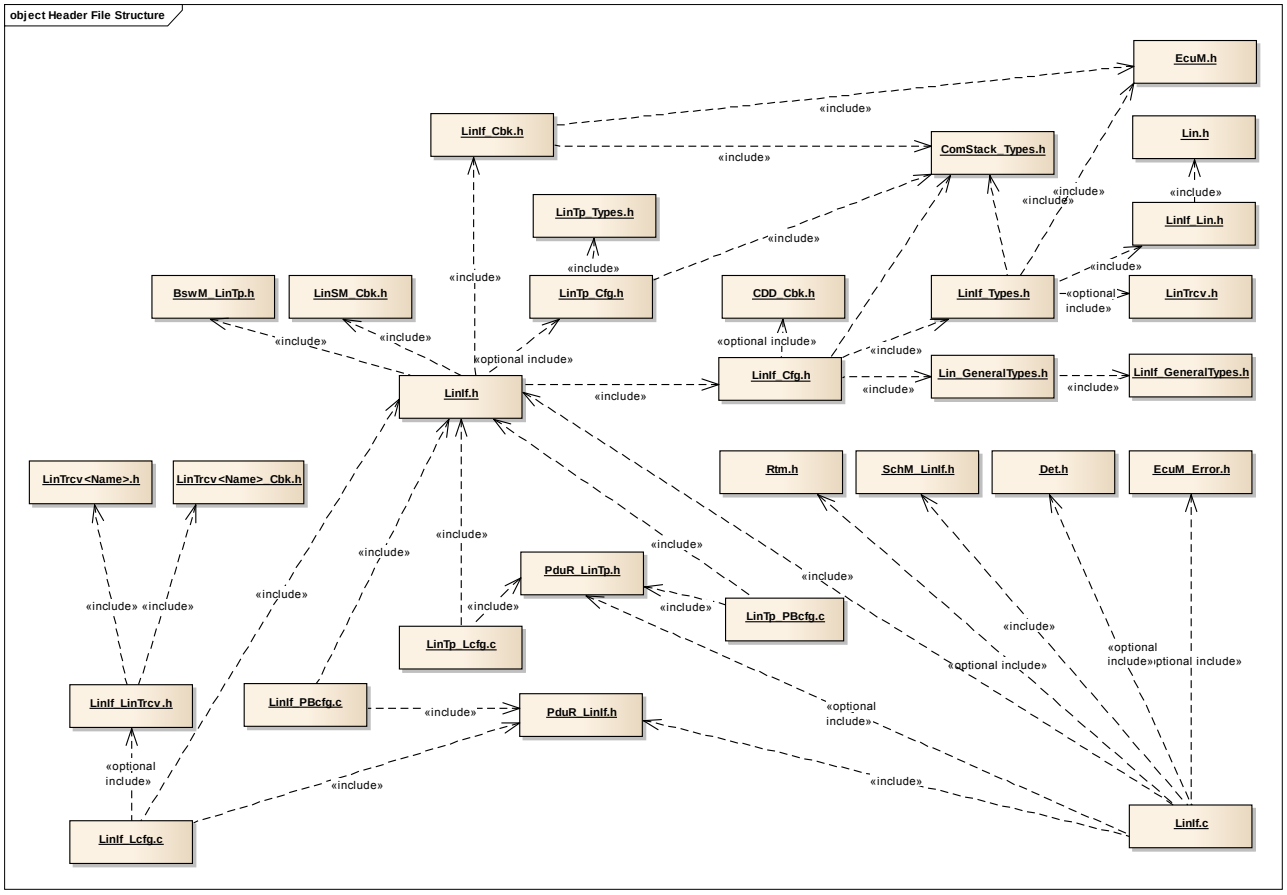

Figure 4-1 Include structure

Optional includes	Description
Det.h	Det.h is only included if reporting to DET is enabled (`LINIF_DEV_ERROR_DETECT == STD_ON`).
EcuM_Error.h	This header file is included if the configuration variant Post-Build-Loadable is active.
LinIf_LinTrcv.h	This header is only generated and included if the LIN Transceiver Driver Handling is enabled.
CDD_Cbk.h	Header files of CDD are only included if CDDs are used as configurable upper layer interfaces.
TP related header	TP header are only included if LinTp support is enabled (`LINIF_TP_SUPPORTED== STD_ON`).

Table 4-4 Optional includes

4.3 Critical Sections

The MICROSAR LIN Interface has no code sections which are executed on interrupt level. But as the LIN interface is calling LIN Driver functions which do not contain any locking mechanism, the LIN interrupts must also be disabled by the LIN Interface interrupt locking.

Furthermore, parts of the main function must not be interrupted by asynchronous task functions of the LinIf to guarantee consistent state transitions.

Critical sections are handled by the BSW Schedule (SchM) [8]. They are automatically configured by the DaVinci Configurator. User interaction is only necessary by updating the internal behavior using the solving action in DaVinci Configurator. It is signaled as a warning in the validaton tab.

The LINIF calls the following function when entering a critical section:

```
SchM_Enter_LinIf_LINIF_EXCLUSIVE_AREA_x()
```

When the critical section is left, the following function is called by LINIF:

```
SchM_Exit_LinIf_LINIF_EXCLUSIVE_AREA_x()
```

4.4 Critical Section Codes

To ensure data consistency and a correct function of the LINIF the two code sections must not be interrupted. Therefore the critical section codes must lead to corresponding interrupt locks, as described below:

LINIF_EXCLUSIVE_AREA_0

- Must lock interrupts of the LIN hardware.
- Must lock global interrupts if API functions of the LINIF can interrupt the `LinIf_MainFunction()` (i.e. preemptive task system, where LinSM is running in higher prior task than LINIF and LinSM is calling `LinIf_GotoSleep()`).

> Must lock interrupts of the other bus communication hardware in case of usage in routing relations (e.g. deactivate CAN interrupts if CAN-LIN-Routing functionality is used).

LINIF_EXCLUSIVE_AREA_1

> Must lock global interrupts if `LinIf_MainFunction()` can interrupt its own API functions (i.e. preemptive task system, where LINIF is running in higher prior task than LinSM and call of `LinIf_GotoSleep()` is interrupted by `LinIf_MainFunction()`).

4.5 Optimizations of ComStackLib

The ComStackLib provides several strategies and algorithms to optimize the required memory footprint. A subset is supported by LINIF and can be optionally enabled in the configuration tool: Inside the Basic editor, right-click the LinIfGeneral node and select in the context menu ‘Create sub container -> LinIfOptimization’ as shown in Figure 4-2:

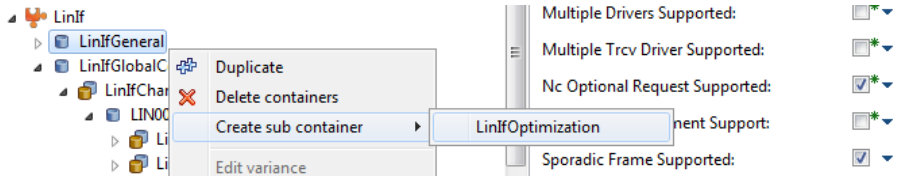

Figure 4-2 Activation of ComStackLib optimizations

The same applies for the LinTpGeneral container. For further information of the particular options, please refer to [11]. By default, all optimization is switched off.

4.6 Integration hints for special use cases

4.6.1 Initial channel state and node management

By default, all LIN channels shall be initialized in SLEEP state to be consistent with other modules of the LIN stack.

For some LIN networks however, it might be the case that no node management is required. This means that communication starts after power-on reset without wakeup frame transmission and the channel also never transits to SLEEP state.

For such channels the startup state can be configured to OPERATIONAL. Furthermore, the sleep support of this channel has to be disabled in the LIN State Manager, see [7] for further details.

4.6.2 LinIf_MainFunction call cycle for UART-based LIN drivers

LIN drivers which are based on UART interfaces work byte-wise, so each transmitted or received byte generates an interrupt. Compared to LIN hardware controller which are able to process a LIN frame completely, such driver implementations require more runtime in

software and are prone to delays in interrupt servicing, e.g. caused by long interrupt locking durations and/or low LIN interrupt priorities. On heavy loaded systems, such delays may result in incomplete LIN frames on the bus.

The actual reception/transmission duration of a LIN frame requires more runtime resources (in the LIN driver) than the residual idle part of a frame slot. `LinIf_MainFunction` triggers the transmission of a LIN frame header at the beginning of a schedule slot, as illustrated in following figure.

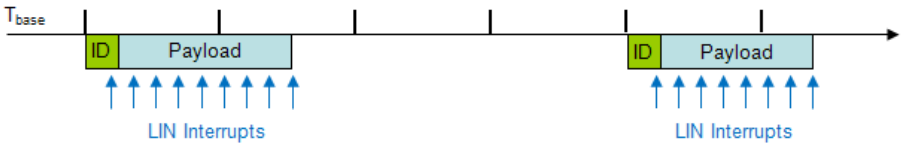

Figure 4-3 LIN frame processing in relation to main function cycles

As consequence, it turned out to be useful to execute `LinIf_MainFunction` asynchronously to other BSW tasks with the same cycle in order to avoid processing LIN frames during peak loads. This can be realized by applying an activation offset to the LinIf_MainFunction task cycle settings.

Reference
If a MICROSAR RTE and OS are used, this setting can be comfortable configured in CFG5, see [10] for further details.

Note
The Load Balancing feature, described in 3.15, is quite related. However, this section addresses the intention to execute the LINIF main function in time slices where the general system capacity is low (only a few other BSWs are executed), while load balancing focuses on the even distribution of the LINIF main function runtimes and jitter minimization.

4.6.3 Synchronization of Schedule Tables

A special requirement for a network might be the synchronous data provision to all slave nodes. E.g. all slave nodes should receive a special command at the same point in time with minimal jitter between each node.

Slave nodes on the same channel are able to receive the same LIN frame without delay because all of them can be configured to handle a common PID, so one frame containing the required information is sufficient.

To achieve synchronicity on a multi-channel configuration requires special handling because the schedule table and frame definitions probably differ on each channel.

Two possible procedures are proposed in order to achieve a simultaneous frame transmission start:

- Possibility 1:
  Preconditions: The time base of all channels is equal. The same schedule table containing exactly one sporadic slot must exist on each channel and this slot delay must match the time base. This also constraints the lengths of the sporadic frame to be transmittable in one cycle / time base tick.
  Procedure: Activate the sporadic schedule table on each channel. In an atomic sequence (e.g. with disabled interrupts) trigger the transmission of all sporadic frames using `LinIf_Transmit`.
- Possibility 2:
  Preconditions: On each channel a schedule table exists which has as first entry an unconditional frame of same length.
  Procedure: Activate the NULL schedule table on all channels. In an atomic sequence (e.g. with disabled interrupts), request the defined schedule table on every channel.

Both routines have the effect that on each channel a specific frame is transmitted at exactly the same cycle. The maximum jitter is given by the runtime of LinIf_MainFunction() to handle all channels.

5. API Description

For an interfaces overview please see Figure 2-2.

5.1 Type Definitions

The types defined by the LINIF are described in this chapter if they are not already described in chapter ‘8 API specification’ of [1] or if they are only used for internal handling.

Type Name	C-Type	Description	Value Range
LinIf_ConfigType	struct	LIN Interface configuration structure	N/A
LinTp_ConfigType	struct	LIN Transport Protocol configuration structure	N/A
LinIf_ScheduleInfoType	struct	LIN Interface schedule information structure	N/A

Table 5-1 Type definitions

5.2 Services provided by LINIF

5.2.1 LinIf_Init

Prototype	
`void LinIf_Init(const void * ConfigPtr)`	
Parameter	
`ConfigPtr`	Pointer to the LIN Interface configuration.
Return code	
-	-
Functional Description	
This function initializes global LIN Interface variables during ECU start-up. It does not initialize modules of lower layer, so e.g.Lin_Init() must be called by ECUM Init Block II before LinIf_Init().	
Particularities and Limitations	
> Has to be called during start-up before LIN communication. > The state of each LinIf channel is set to either NORMAL (operational) or SLEEP state, depending on the configuration of ‘LinIfStartupState’ feature.	
Call context	
> Global interrupts must be disabled. > Must only be called once during run time.	

Table 5-2 LinIf_Init

5.2.2 LinIf_InitMemory

Prototype	
`void LinIf_InitMemory(void)`	
Parameter	
-	-
Return code	
-	-
Functional Description	
This function initializes memory of LINIF (as well as of LinTp) so that expected startup values are set.	
Particularities and Limitations	
> If this function is used, it must be called before any other service function of the LIN interface, including LinIf_Init().	
Call context	
> Global interrupts must be disabled. > Must only be called once during run time.	

Table 5-3 LinIf_InitMemory

5.2.3 LinIf_GetVersionInfo

Prototype	
`void LinIf_GetVersionInfo(Std_VersionInfoType *versioninfo)`	
Parameter	
`versioninfo`	Pointer to which the version information of this module is written.
Return code	
–	-
Functional Description	
This service returns version information, vendor ID and AUTOSAR module ID of the component. The versions are decimal-coded.	
Particularities and Limitations	
> The function is only available if enabled by parameter ‘LinIfVersionInfoApi’ (LINIF_VERSION_INFO_API == ON).	
Call context	
> Can always be called.	

Table 5-4 LinIf_GetVersion

5.2.4 LinIf_Transmit

Prototype	
`Std_ReturnType LinIf_Transmit(PduIdType LinTxPduId, const PduInfoType *PduInfoPtr)`	
Parameter	
`LinTxPduId` `PduInfoPtr`	Upper layer identification of the LIN frame to be transmitted. Unused pointer to a structure with frame related data.
Return code	
`Std_ReturnType`	Result of function can be: `E_OK, E_NOT_OK`
Functional Description	
A call of this function indicates a request from an upper layer to transmit the frame specified by the LinTxPduId. This call only marks a sporadic frame as pending for transmission. Invocation for non-sporadic frame is refused (meaningless). Repeated invocations while the sporadic frame is still pending are tolerated, but the PDU will only be set to pending once.	
Particularities and Limitations	
> This function contains only code if sporadic frame support is enabled in the generation tool. The return value is always E_NOT_OK if disabled.	
Call context	
> Can always be called after initialization of module.	

Table 5-5 LinIf_Transmit

5.2.5 LinIf_ScheduleRequest

Prototype	
`Std_ReturnType LinIf_ScheduleRequest(NetworkHandleType Channel, LinIf_SchHandleType Schedule)`	
Parameter	
`Channel` `Schedule`	The channel on which a schedule table request shall be done. Identification of the new schedule to be set.
Return code	
`Std_ReturnType`	Result of function can be: `E_OK, E_NOT_OK`
Functional Description	
This function can be used by different upper layers to request a schedule table to be executed.	
Particularities and Limitations	
> If any not existing table is referenced or the channel is in SLEEP, the function returns with E_NOT_OK. > Usage of exclusive areas.	
Call context	
> Can always be called after initialization of module.	

Table 5-6 LinIf_ScheduleRequest

5.2.6 LinIf_GotoSleep

Prototype	
`Std_ReturnType LinIf_GotoSleep(NetworkHandleType Channel)`	
Parameter	
`Channel`	The channel on which the enter sleep mode request shall be done.
Return code	
`Std_ReturnType`	Result of function can be: `E_OK, E_NOT_OK`
Functional Description	
This call initiates a transition into sleep mode on the selected channel. If the channel is not in wake state, this call does nothing and returns `E_NOT_OK`. The transition is carried out by transmitting a Master Request frame with its first data byte equal to 0x00. This function will start the process of putting the cluster into sleep, where the sleep mode frame is not triggered immediately. It will be started as soon as the current slot is due. This call has no effect if the LIN channel is already serving the go-to-sleep.	
Particularities and Limitations	
> Usage of exclusive areas.	
Call context	
> Can always be called after initialization of module.	

Table 5-7 LinIf_GotoSleep

5.2.7 LinIf_Wakeup

Prototype	
`Std_ReturnType LinIf_Wakeup(NetworkHandleType Channel)`	
Parameter	
`Channel`	The channel on which the enter wake mode request shall be done.
Return code	
`Std_ReturnType`	Result of function can be: `E_OK, E_NOT_OK`
Functional Description	
This call initiates the wake up process on the selected channel.	
Particularities and Limitations	
> Usage of exclusive areas.	
Call context	
> Can always be called after initialization of module.	

Table 5-8 LinIf_Wakeup

5.2.8 LinIf_SetTrcvMode

Prototype	
`Std_ReturnType LinIf_SetTrcvMode(NetworkHandleType Channel, LinTrcv_TrcvModeType TransceiverMode)`	
Parameter	
`Channel`	The channel on which to set the new transceiver state.
`TransceiverMode`	The new transceiver mode to set
Return code	
`Std_ReturnType`	Result of function can be: `E_OK, E_NOT_OK`
Functional Description	
This call requests the underlying transceiver to change the state into the new mode.	
Particularities and Limitations	
> None.	
Call context	
> Can always be called after initialization of module.	

Table 5-9 LinIf_SetTrcvMode

5.2.9 LinIf_GetTrcvMode

Prototype	
`Std_ReturnType LinIf_GetTrcvMode(NetworkHandleType Channel, LinTrcv_TrcvModeType* TransceiverModePtr)`	
Parameter	
`Channel`	The channel on which to request the current transceiver mode.
`TransceiverModePtr`	Pointer to memory location where the mode will be set.
Return code	
`Std_ReturnType`	Result of function can be: `E_OK, E_NOT_OK`
Functional Description	
This call requests the current mode of the underlying transceiver.	
Particularities and Limitations	
> None.	
Call context	
> Can always be called after initialization of module.	

Table 5-10 LinIf_GetTrcvMode

© 2017 Vector Informatik GmbH

5.2.10 LinIf_GetTrcvWakeupReason

Prototype	
`Std_ReturnType LinIf_GetTrcvWakeupReason(NetworkHandleType Channel, LinTrcv_TrcvWakeupReasonType* TrcvWuReasonPtr)`	
Parameter	
`Channel`	The channel on which to request the wakeup reason.
`TrcvWuReasonPtr`	Pointer to memory location where the wakeup reason to store.
Return code	
`Std_ReturnType`	Result of function can be: `E_OK, E_NOT_OK`
Functional Description	
This call returns the wakeup reason that has been detected by the underlying LIN transceiver. In case the transceiver is not in normal mode, the development error LINIF_E_TRCV_NOT_NORMAL is reported and E_NOT_OK is returned.	
Particularities and Limitations	
> None.	
Call context	
> Can always be called after initialization of module.	

Table 5-11 LinIf_GetTrcvWakeupReason

5.2.11 LinIf_SetTrcvWakeupMode

Prototype	
`Std_ReturnType LinIf_SetTrcvWakeupMode(NetworkHandleType Channel, LinTrcv_TrcvWakeupModeType LinTrcvWakeupMode)`	
Parameter	
`Channel`	The channel on which to set the wakeup notification state.
`LinTrcvWakeupMode`	The new state for wakeup event notification of the transceiver.
Return code	
`Std_ReturnType`	Result of function can be: `E_OK, E_NOT_OK`
Functional Description	
This call enables, disables or clears the notification for wakeup events.	
Particularities and Limitations	
> None.	
Call context	
> Can always be called after initialization of module.	

Table 5-12 LinIf_SetTrcvWakeupMode

5.2.12 LinIf_CancelTransmit

Prototype	
`Std_ReturnType LinIf_CancelTransmit(PduIdType LinTxPduId)`	
Parameter	
`LinTxPduId`	Identification of the LIN frame for which a cancellation should be done.
Return code	
`Std_ReturnType`	Result of function can be: `E_OK`
Functional Description	
This function does nothing, it's only purpose is interface compatibility. Returns always E_OK.	
Particularities and Limitations	
> None.	
Call context	
> Can always be called after initialization of module.	

Table 5-13 LinIf_CancelTransmit

5.2.13 LinIf_CheckWakeup

Prototype	
`Std_ReturnType LinIf_CheckWakeup(EcuM_WakeupSourceType WakeupSource)`	
Parameter	
`WakeupSource`	Source which initiated the wakeup event. May be either LIN driver or LIN transceiver.
Return code	
`Std_ReturnTyoe`	Result of function can be: `E_OK, E_NOT_OK`
Functional Description	
This function is called by EcuM when LIN driver or LIN transceiver has detected a wakeup on the LIN line and notified upper layer about this event. The LIN interface converts the wakeup source to the corresponding channel and forward the call to the lower layer by either calling `Lin_CheckWakeup()` of the underlying LIN driver or by calling `LinTrcv_CheckWakeup()` of the LIN transceiver if available.	
Particularities and Limitations	
> Usage of exclusive areas. > This function is only available if wakeup support is enabled on at least one LIN channel.	
Call context	
> Can always be called after initialization of module.	

Table 5-14 LinIf_ CheckWakeup

5.2.14 LinIf_MainFunction

Prototype
`void LinIf_MainFunction(void)`

Parameter	
-	-
Return code	
-	-
Functional Description	
This function has to be called each tick (fixed cyclic). Fixed cyclic means that one cycle time is defined at configuration and shall not be changed because functionality is requiring that fixed timing. For further information about the call cycle refer to chapter 3.6 or to chapter 7.2.1 in [1].	
Particularities and Limitations	
> Usage of exclusive areas for each channel specific handling.	
Call context	
> Can always be called after initialization of module.	

Table 5-15 LinIf_MainFunction

5.2.15 LinIf_GetScheduleInfo

Prototype	
`void LinIf_GetScheduleInfo(NetworkHandleType Channel, LinIf_ScheduleInfoType ScheduleInfo)`	
Parameter	
`Channel` `ScheduleInfo`	The channel on which to request the schedule table information. Struct pointer to store the schedule information to.
Return code	
-	-
Functional Description	
This function returns the current state of the schedule table handler. The returned information contains the current table identifier, current slot index, requested (pending) table identifier, current slot length and position inside slot.	
Particularities and Limitations	
> Usage of exclusive areas. > This function is only available if schedule info API is enabled.	
Call context	
> Can always be called after initialization of module.	

Table 5-16 LinIf_GetScheduleInfo

5.2.16 LinTp_Init

Prototype	
`void LinTp_Init(const LinTp_ConfigType * ConfigPtr)`	
Parameter	
`ConfigPtr`	Pointer to the LIN TP configuration.

Return code	
–	-
Functional Description	
This function initializes global LIN TP variables during ECU start-up. The connection state of each TP channel is set to idle state.	
Particularities and Limitations	
> It is recommended to init the TP after the LinIf initialization	
Call context	
> Global interrupts must be disabled. > Must only be called once during run time.	

Table 5-17 LinTp_Init

5.2.17 LinTp_Transmit

Prototype	
`Std_ReturnType LinTp_Transmit(PduIdType LinTpTxSduId, const PduInfoType * LinTpTxInfoPtr)`	
Parameter	
`LinTpTxSduId`	This parameter contains the unique identifier of the NSDU (TP message) to be transmitted.
`LinTpTxInfoPtr`	A pointer to a structure with N-SDU related data
Return code	
`Std_ReturnType`	Result of function can be: `E_OK, E_NOT_OK`
Functional Description	
This service is used to request the transfer of segmented data over the LIN bus.	
Particularities and Limitations	
> Usage of exclusive areas.	
Call context	
> Can always be called after initialization of module.	

Table 5-18 LinTp_Transmit

5.2.18 LinTp_GetVersionInfo

Prototype	
`void LinTp_GetVersionInfo(Std_VersionInfoType * versioninfo)`	
Parameter	
`Versioninfo`	Struct pointer for version info storing
Return code	
–	-

Functional Description
This service returns version information, vendor ID and AUTOSAR module ID of the component. The versions are decimal-coded.
Particularities and Limitations
> None.
Call context
> The function is only available if enabled at build time (LINTP_VERSION_INFO_API == ON).

Table 5-19 LinTp_GetVersionInfo

5.2.19 LinTp_Shutdown

Prototype	
`void LinTp_Shutdown(void)`	
Parameter	
–	-
Return code	
–	-
Functional Description	
This service closes all pending transport protocol connection, frees all resources and sets the corresponding LinTp module into the LINTP_UNINIT state.	
Particularities and Limitations	
> The is no notification to upper layers on pending connection (PduR)	
Call context	
> Can always be called after initialization of module.	

Table 5-20 LinTp_Shutdown

5.2.20 LinTp_ChangeParameter

Prototype	
`Std_ReturnType LinTp_ChangeParameter(PduIdType id, TPParameterType parameter, uint16 value)`	
Parameter	
`id`	Identifier of the received N_SDU on which the reception parameter has to be changed.
`parameter`	The selected parameter that the request shall change (STmin).
`value`	The new value of the parameter.
Return code	
`Std_ReturnType`	Result of function is always: `E_NOT_OK (requested by standard)`
Functional Description	
This function does nothing, it's is only for interface compatibility. The function is only available if the corresponding feature in the PduR module is enabled.	

Particularities and Limitations
> None.
Call context
> Can always be called after initialization of module.

Table 5-21 LinTp_ChangeParameter

5.2.21 LinTp_CancelTransmit

Prototype	
`Std_ReturnType LinTp_CancelTransmit (PduIdType LinTpTxSduId)`	
Parameter	
`LinTpTxSduId`	This parameter contains the Lin TP instance unique identifier of the Lin N-SDU which transfer has to be cancelled.
Return code	
`Std_ReturnType`	Result of function is always: `E_NOT_OK (requested by standard)`
Functional Description	
This function does nothing, it's is only for interface compatibility. The function is only available if the corresponding feature in the PduR module is enabled.	
Particularities and Limitations	
> This function is only available if configuration parameter 'Cancel Transmit Supported' is enabled.	
Call context	
> Can always be called after initialization of module.	

Table 5-22 LinTp_CancelTransmit

5.2.22 LinTp_CancelReceive

Prototype	
`Std_ReturnType LinTp_CancelReceive (PduIdType LinTpRxSduId)`	
Parameter	
`LinTpRxSduId`	This parameter contains the Lin TP instance unique identifier of the Lin N-SDU which reception has to be cancelled.
Return code	
`Std_ReturnType`	Result of function is always: `E_NOT_OK (requested by standard)`
Functional Description	
This function does nothing, it's is only for interface compatibility. The function is only available if the corresponding feature in the PduR module is enabled.	
Particularities and Limitations	
> This function is only available if configuration parameter 'Cancel Receive Supported' is enabled.	

Call context
> Can always be called after initialization of module.

Table 5-23 LinTp_CancelReceive

5.3 Services used by LINIF

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

Component	API
DET	Det_ReportError()
EcuM	EcuM_GeneratorCompatibilityError()
SchM	SchM_Enter_LinIf_LINIF_EXCLUSIVE_AREA_0() SchM_Exit_LinIf_LINIF_EXCLUSIVE_AREA_0() SchM_Enter_LinIf_LINIF_EXCLUSIVE_AREA_1() SchM_Exit_LinIf_LINIF_EXCLUSIVE_AREA_1()
Lin	Lin_SendFrame() Lin_GetStatus Lin_GoToSleep() Lin_GoToSleepInternal() Lin_Wakeup() Lin_WakeupInternal() Lin_CheckWakeup()
BswM	BswM_LinTp_RequestMode()
LinTrcv	LinTrcv_CheckWakeup() LinTrcv_GetOpMode() LinTrcv_SetOpMode() LinTrcv_GetBusWuReason() LinTrcv_SetWakeupMode()
PduR	PduR_LinTpCopyRxData() PduR_LinTpCopyTxData() PduR_LinTpRxIndication() PduR_LinTpStartOfReception() PduR_LinTpTxConfirmation()
Rtm	Rtm_Start() Rtm_Stop()
LinSm	LinSM_ScheduleEndNotification()

Table 5-24 Services used by the LINIF

5.4 Callback Functions

This chapter describes the callback functions that are implemented by the LINIF and can be invoked by other modules. The prototypes of the callback functions are provided in the header file LinIf_Cbk.h.

5.4.1 LinIf_WakeupConfirmation

Prototype	
`void LinIf_WakeupConfirmation (EcuM_WakeupSourceType WakeupSource)`	
Parameter	
`WakeupSource`	This parameter contains the source which initiated the wakeup event. May be either LIN driver or LIN transceiver.
Return code	
–	-
Functional Description	
The LIN driver or LIN transceiver will call this function to report the wakeup source after the successful wakeup detection during CheckWakeup or after power on by bus.	
Particularities and Limitations	
> None.	
Call context	
> Can always be called after initialization of module.	

Table 5-25 LinIf_WakeupConfirmation

5.5 Configurable Interfaces

5.5.1 Notifications

At its configurable interfaces the LINIF defines notifications that can be mapped to callback functions provided by other modules. The mapping is not statically defined by the LINIF but can be performed at configuration time. The function prototypes that can be used for the configuration have to match the appropriate function prototype signatures, which are described in the following sub-chapters.

5.5.1.1 <User>_ScheduleRequestConfirmation

Prototype	
`void <User>_ScheduleRequestConfirmation (NetworkHandleType channel, LinIf_SchhandleType schedule)`	
Parameter	
`channel` `schedule`	The channel on which a schedule table change was performed. The new, currently active schedule table.
Return code	
–	-
Functional Description	
This function is called after a schedule table change request via **`LinIf_ScheduleRequest()`** was performed successfully. The schedule table given by parameter *schedule* is already active.	

Particularities and Limitations
> The function to be called is configured for each channel by parameters LinIfScheduleRequestConfirmationUL and LinIfScheduleRequestConfirmationFctUL. If LinIfScheduleRequestConfirmationUL is set to LINSM, the callback function is fixed to `LinSM_ScheduleRequestConfirmation()`. If LinIfScheduleRequestConfirmationUL is set to CDD, the callback function name has to be configured via parameter LinIfScheduleRequestConfirmationFctUL.
Call context
> Called on task level outside exclusive areas.

Table 5-26 <User>_ScheduleRequestConfirmation

5.5.1.2 <User>_GotoSleepConfirmation

Prototype	
`void <User>_GotoSleepConfirmation (NetworkHandleType channel, boolean success)`	
Parameter	
`channel`	The channel which has transited to SLEEP state.
`success`	TRUE if a goto sleep frame was sent on the bus, otherwise FALSE.
Return code	
–	-
Functional Description	
This function is called after a channel has performed the transition to SLEEP state after a **`LinIf_GotoSleep()`** request. If this function is called, the channel has entered SLEEP state already, independent of parameter *success*.	
Particularities and Limitations	
> The function to be called is configured for each channel by parameters LinIfGotoSleepConfirmationUL and LinIfGotoSleepConfirmationFctUL. If LinIfGotoSleepConfirmationUL is set to LINSM, the callback function is fixed to `LinSM_GotoSleepConfirmation()`. If LinIfGotoSleepConfirmationUL is set to CDD, the callback function name has to be configured via parameter LinIfGotoSleepConfirmationFctUL.	
Call context	
> Called on task level outside exclusive areas.	

Table 5-27 <User>_GotoSleepConfirmation

5.5.1.3 <User>_WakeupConfirmation

Prototype	
`void <User>_WakeupConfirmation (NetworkHandleType channel, boolean success)`	
Parameter	
`channel`	The channel which has transited to OPERATIONAL state.
`success`	TRUE if a wakeup transition was successful, otherwise FALSE.

Return code	
–	-
Functional Description	
This function is called after a transition to OPERATIONAL state by a **`LinIf_Wakeup()`** call was requested. If this function is called with *success* parameter TRUE, the channel has entered OPERATIONAL state. If it's called with *success* parameter FALSE, the request was not accepted. This can only happen due to a misconfiguration, in this case LinIf_Wakeup() has already returned E_NOT_OK.	
Particularities and Limitations	
> The function to be called is configured for each channel by parameters LinIfWakeupConfirmationUL and LinIfWakeupConfirmationFctUL. If LinIfWakeupConfirmationUL is set to LINSM, the callback function is fixed to `LinSM_WakeupConfirmation()`. If LinIfWakeupConfirmationUL is set to CDD, the callback function name has to be configured via parameter LinIfWakeupConfirmationFctUL.	
Call context	
> Called on task level outside exclusive areas.	

Table 5-28 <User>_WakeupConfirmation

5.5.1.4 <User>_TriggerTransmit

Prototype	
`Std_ReturnType <User>_TriggerTransmit (PduIdType` `TxPduId`**`, PduInfoType*`** `PduInfoPtr`**`)`**	
Parameter	
`TxPduId`	ID of the SDU that is request to be transmitted.
`PduInfoPtr`	Pointer to a PDU info structure which has to be filled with the SDU data to be transmitted and with the SDU length.
Return code	
`Std_ReturnType`	Result of function can be: `E_OK, E_NOT_OK`
Functional Description	
This function is called by LINIF before transmission of a TX frame. The *PduInfoPtr* structure points to a valid buffer which the upper layer has to fill with transmission data.	
Particularities and Limitations	
> The function to be called is configured for each TxPdu by parameters LinIfUserTxUL and LinIfTxTriggerTransmitUL. If LinIfUserTxUL is set to PDUR, the callback function is fixed to `PduR_LinIfTriggerTransmit()`. If LinIfUserTxUL is set to CDD, the callback function name has to be configured via parameter LinIfTxTriggerTransmitUL.	
Call context	
> Called on task level outside exclusive areas.	

Table 5-29 <User>_TriggerTransmit

5.5.1.5 <User>_TxConfirmation

Prototype	
`void <User>_TxConfirmation (PduIdType TxPduId)`	
Parameter	
`TxPduId`	ID of the SDU that has been transmitted.
Return code	
–	-
Functional Description	
This function is called by LINIF after transmission of a TX frame. It is not called if transmission has failed.	
Particularities and Limitations	
> The function to be called is configured for each TxPdu by parameters LinIfUserTxUL and LinIfTxConfirmationUL. If LinIfUserTxUL is set to PDUR, the callback function is fixed to `PduR_LinIfTxConfirmation()`. If LinIfUserTxUL is set to CDD, the callback function name has to be configured via parameter LinIfTxConfirmationUL.	
Call context	
> Called on task level outside exclusive areas.	

Table 5-30 <User>_TxConfirmation

5.5.1.6 <User>_RxIndication

Prototype	
`void <User>_RxIndication (PduIdType RxPduId, PduInfoType* PduInfoPtr)`	
Parameter	
`TxPduId` `PduInfoPtr`	ID of the SDU that has been received. Pointer to a Pdu structure containing the received data and length information.
Return code	
–	-
Functional Description	
This function is called by LINIF after having received a RX frame successfully.	
Particularities and Limitations	
> The function to be called is configured for each RxPdu by parameters LinIfUserRxIndicationUL and LinIfRxIndicationUL. If LinIfUserRxIndicationUL is set to PDUR, the callback function is fixed to `PduR_LinIfRxIndication()`. If LinIfUserRxIndicationUL is set to CDD, the callback function name has to be configured via parameter LinIfRxIndicationUL.	
Call context	
> Called on task level outside exclusive areas.	

Table 5-31 <User>_RxIndication

5.6 Callouts

5.6.1 Appl_LinIfGetLinStatus

Prototype	
`void Appl_LinIfGetLinStatus (NetworkHandleType Channel, Lin_FramePidType Pid, Lin_StatusType Status)`	
Parameter	
`Channel`	The channel on which a frame was processed.
`Pid`	Protected identifier of processed frame.
`Status`	Lin driver status for processed frame.
Return code	
–	-
Functional Description	
This callout function is called after each frame to inform the upper layer directly about the status returned by LIN driver of the processed frame.	
Particularities and Limitations	
> This callout function has to be enabled by specifying a user config file with following content: `#define LINIF_APPL_FRAME_STATUS_INFO STD_ON`	
Call context	
> Called on task level inside exclusive areas.	

Table 5-32 Appl_LinIfGetLinStatus

6. Configuration

In the LINIF the attributes can be configured according to/ with the following methods/ tools:

- Configuration in DaVinci Configurator 5

6.1 Configuration Variants

The LINIF supports the configuration variants

- VARIANT-PRE-COMPILE
- VARIANT-POSTBUILD-LOADABLE
- VARIANT-POSTBUILD-SELECTABLE

The configuration classes of the LINIF parameters depend on the supported configuration variants. For their definitions please see the LinIf_bswmd.arxml file.

7. AUTOSAR Standard Compliance

7.1.1 Deviations within Configuration Parameters for Lin Interface

Configuration Parameter	Deviation	Reason
Wakeup Delay	Added	No wakeup delay had been specified after an internal wakeup request. According to [6] the default delay is 100ms, but can be less. This value can be configured.
Wakeup Delay External	Added	No wakeup delay had been specified after an external wakeup was detected. According to [6] the default delay is 100ms, but can be less. This value can be configured.
Support Sporadic Frame Handling	Added	Code and runtime optimization in case sporadic frames are not used.
Support Event Triggered Frame Handling	Added	Code and runtime optimization in case sporadic frames are not used.
Configurable function names for upper layer interfaces on channel level	Added	Flexible functions names for CDD usage.
Optional Node Configuration Requests (LinIfNcOptionalRequest Supported)	Unused	Optional NAD services are always supported by Vector LINIF if configured as schedule table commands.
LinIfTrcvWakeupNotification	Removed	Parameter is unused and removed according to AR Bugzilla #52901 (AR4-569)
LinIfSlave container	Range changed	Multiplicity changed from 0..* to 0..16 according to LIN 2.x specification.
Load Balancing offset	Added	Support of channel-based load balancing feature.
Schedule End Notification	Added	Support for notification callback to LinSM if current schedule table is run through.
SAE J2602 frame tolerance support	Added	Four parameters are added: - General J2602 frame tolerance support switch - Header tolerance of master for each LinIf channel - Response tolerance of master for each LinIf channel - Response tolerance for each Rx frame
Safe Bsw Checks	Added	Support for component-specific SafeBSW configuration.
Schedule Info Api	Added	Activation of optional Schedule Information API.
Schedule Change Next TimeBase	Added	According to AUTOSAR 4.3.0. Configuration of point in time of actual schedule table switch

7.1.2 Deviations within Configuration Parameters for Lin Transport Protocol

Forward Response Pending Frames	Added	The feature of not forwarding response pending frames to PduR is a requested customer option.
Functional Requests Supported	Added	Code optimization in case functional requests are not required.
LinIf channel reference	Added	Required for the connection between LinTp channel options and the actual LinIf channel
P2/P2Max	Range changed	The minimum and maximum values of both parameters are changed according to AR Bugzilla #52900 (AR4-569) and coincide with AUTOSAR 4.1.
Broadcast Request Handling	Added	Support of extended handling of diagnostic broadcast requests.

7.1.3 Deviations within API

API	Deviation	Reason
LinIf_InitMemory()	Additional API	Some compiler / startup codes do not support initialization of variables. This can now also be done by calling this function before calling the initialization function.
LinSM_ScheduleEndNotification()	Additional API	Optional API to allow upper layer to change schedule table if the current one is processed completely.
LinIf_GetScheduleInfo()	Additional API	Optional API to request the current state of the schedule table manager.
Appl_LinIfGetLinStatus()	Additional API	Optional API to inform the upper layer about the LIN driver status for each frame slot.

7.1.4 Deviations within features

Feature	Deviation	Reason
Multiple LIN Driver	Not supported	Multiple LIN drivers are not supported.
TP reception handling	Changed	Received TP frames with invalid PCI or length do not lead to reception abortion, but are ignored to be compliant to [6].
Collision Resolving	Limited	A RUN_ONCE schedule table is not allowed to have more than one EVT frame (slot) for correct collision resolving.

LinTp Buffer Handling	Changed	The algorithm for buffer requests from the PduR is implemented according to AUTOSAR 4.1.1. Therefore the return value BUFREQ_E_BUSY of AUTOSAR 4.0.3 is not used. For further details, see [9].
Wakeup Confirmation	Changed	The wakeup confirmation is adapted to AR Bugzilla #57401 (AR4-612) and coincides with AUTOSAR 4.1.x.
File structure	Changed	Additional files have been introduced for the implementation of the LIN Interface (s. 4.2). None of the additional introduced files must be included or handled by any other modules, so no adaptations to other modules are necessary.
Interface between PduR and LinTp	Changed	According to ASR4.1.2 (AR4-619): - PduR_LinTpStartOfReception API: new info parameter of type PduInfoType - PduR_LinTpTxConfirmation and PduR_LinTpRxIndication API: result parameter type changed to Std_ReturnType
SAE J2602 frame tolerance	Added	According to SAE J2602, a maximum frame tolerance different from 40% of the LIN standard is supported.

8. Glossary and Abbreviations

8.1 Glossary

Term	Description
CFG5	DaVinci Configurator 5, generation tool for MICROSAR components

Table 8-1 Glossary

8.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
BSWM	BSW ModeManager
CF	Consecutive Frame
COM	AUTOSAR Communication Module
DEM	Diagnostic Event Manager
DET	Development Error Tracer
ECU	Electronic Control Unit
ECUM	ECU State Manager
EVT Frame	Event-Triggered Frame
FF	First Frame
HIS	Hersteller Initiative Software
ISR	Interrupt Service Routine
LINSM	LIN State Manager
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
MRF	Master Request Frame
PDUR	PDU Router
RTE	Runtime Environment
RTM	Runtime Measurement
SAE	Society of Automobile Engineers
SF	Single Frame
SID	Service Identifier
SRF	Slave Response Frame
SRS	Software Requirement Specification
SWC	Software Component
SWS	Software Specification
TP	Transport Protocol

Table 8-2 Abbreviations

9. Contact

Visit our website for more information on

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

www.vector.com