MICROSAR Classic LIN Interface

Technical Reference

AUTOSAR 4

Version 10.00.00

Authors	visbmo, vispwf
Status	Released

Document Information

History

Author	Date	Version	Remarks
visbmo	2012-07-16	1.00.00	Initial version
visbmo	2012-11-08	1.00.01	Corrected misspellings
visbmo	2013-03-05	1.01.00	Added Post-build support, removed unused Vector parameters
visbmo	2013-09-30	2.00.00	Improved several chapters, removed redundant description of BSWMD parameters, added Runtime measurement
visbmo	2014-01-07	2.01.00	Added AR4-569 changes, adapted architecture and TP timeout figures
visbmo	2014-05-05	3.00.00	Removed BRS upper layer parameter Adapted wakeup confirmation handling LinTp/PduR interface conform to AR4.1.2
visbmo	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
visbmo	2015-01-08	4.01.00	Added schedule end notification
visbmo	2015-06-18	4.02.00	Added diagnostic communication chapter
visbmo	2016-02-08	4.03.00	Added SAE J2602 frame tolerance support
visbmo	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
visbmo	2017-11-20	4.04.01	Removed incorrect deviations about not-supported multiple driver support
visbmo	2018-08-13	5.00.00	Added bus mirroring
visbmo	2019-12-30	6.00.00	Added LIN slave support Removed obsolete note container
visbmo	2020-03-11	6.00.01	Added corrections from review findings
visbmo	2020-06-05	6.01.00	Added wakeup confirmation timeout
visbmo	2020-07-15	7.00.00	Added SAE J2602 Targeted Reset support Added SAE J2602 Status byte handling
visbmo	2020-12-04	7.01.00	Added LIN master jitter consideration
visbmo	2021-01-21	7.02.00	Added ReadByIdentifier support for optional identifiers (LIN slave) Added DataDump and AutoAddressing support (LIN slave)

visbmo	2021-06-18	7.03.00	Added meta data handling for LinTp
visbmo	2022-02-04	8.00.00	Added channel based main functions Removed LoadBalancing
vispwf	2022-06-25	8.01.00	Added DET runtime error reporting
visbmo	2023-09-05	9.00.00	Added support of 3rd party LIN driver compliant to AUTOSAR 4.3.1: - Configurable LIN driver startup state - Configurable call to Lin_GetStatus after wakeup request - Usage of legacy frame response type - Harmonize internal sleep transition
vispwf	2024-01-29	10.00.00	Added support of 3rd party LIN driver compliant to AUTOSAR 4.4.0 and 21-11 - Configurable channel handle for LIN driver callback functions - Update include structure

Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	Specification of LIN Interface	R4.4.0
[2]	AUTOSAR	Specification of Default Error Tracer	R4.4.0
[3]	AUTOSAR	List of Basic Software Modules	R4.4.0
[4]	AUTOSAR	Specification of Basic Software Mode Manager	R4.4.0
[5]	LIN	LIN Specification Package, Revision 2.1, November 24, 2006	2.1
[6]	ISO	ISO 17987-2 Road vehicles — Local Interconnect Network (LIN) — Part 2: Transport protocol and network layer services	2016
[7]	ISO	ISO 17987-3 Road vehicles — Local Interconnect Network (LIN) — Part 3: Protocol specification	2016
[8]	Vector	MICROSAR Classic LIN State Manager Technical Reference	see delivery
[9]	AUTOSAR	Specification of RTE Software	R4.4.0
[10]	Vector	MICROSAR Classic PDU Router Technical Reference	see delivery
[11]	Vector	MICROSAR Classic RTE Technical Reference	see delivery
[12]	Vector	MICROSAR Classic ComStackLib Technical Reference	see delivery
[13]	Vector	MICROSAR Classic COM Technical Reference	see delivery
[14]	Vector	MICROSAR Classic Complex Device Driver	see delivery
[15]	SAE	SAE J2602-1 NOV2012	2012

Scope of the Document

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

The term LINIF is not strictly used for the LIN Interface, but also includes the LIN Transport Protocol. The term LINTP may also be used for LIN Transport Protocol properties.

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.

Contents

Illustrations

Tables

1 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 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. [3])

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

1.1 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 Default Error Tracer
BswM_	Services of Basic Software Mode Manager
EcuM_	Services of ECU State Manager
Com_	Services of Communication module

Table 1-1 Naming Conventions

The term “(LIN master)” denotes functionality that is only applicable to LIN channels that are configured as LIN master nodes. Similar, the term “(LIN slave)” denotes functionality that is only applicable to LIN channels that are configured as LIN slave nodes.

1.2 Architecture Overview

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

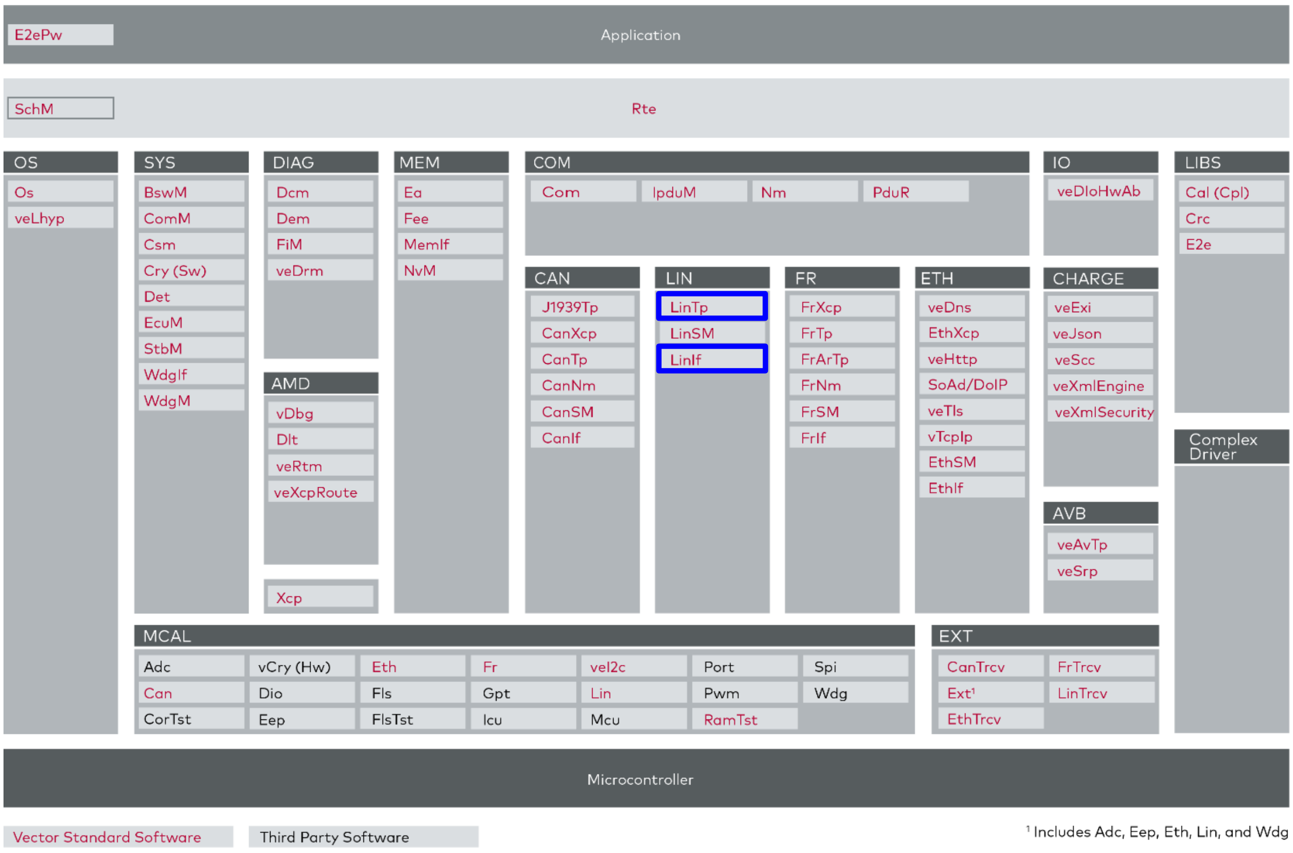


Figure 1-1 AUTOSAR 4.x Architecture Overview

The next figure shows the interfaces to adjacent modules of the LINIF. These interfaces are described in chapter 4.

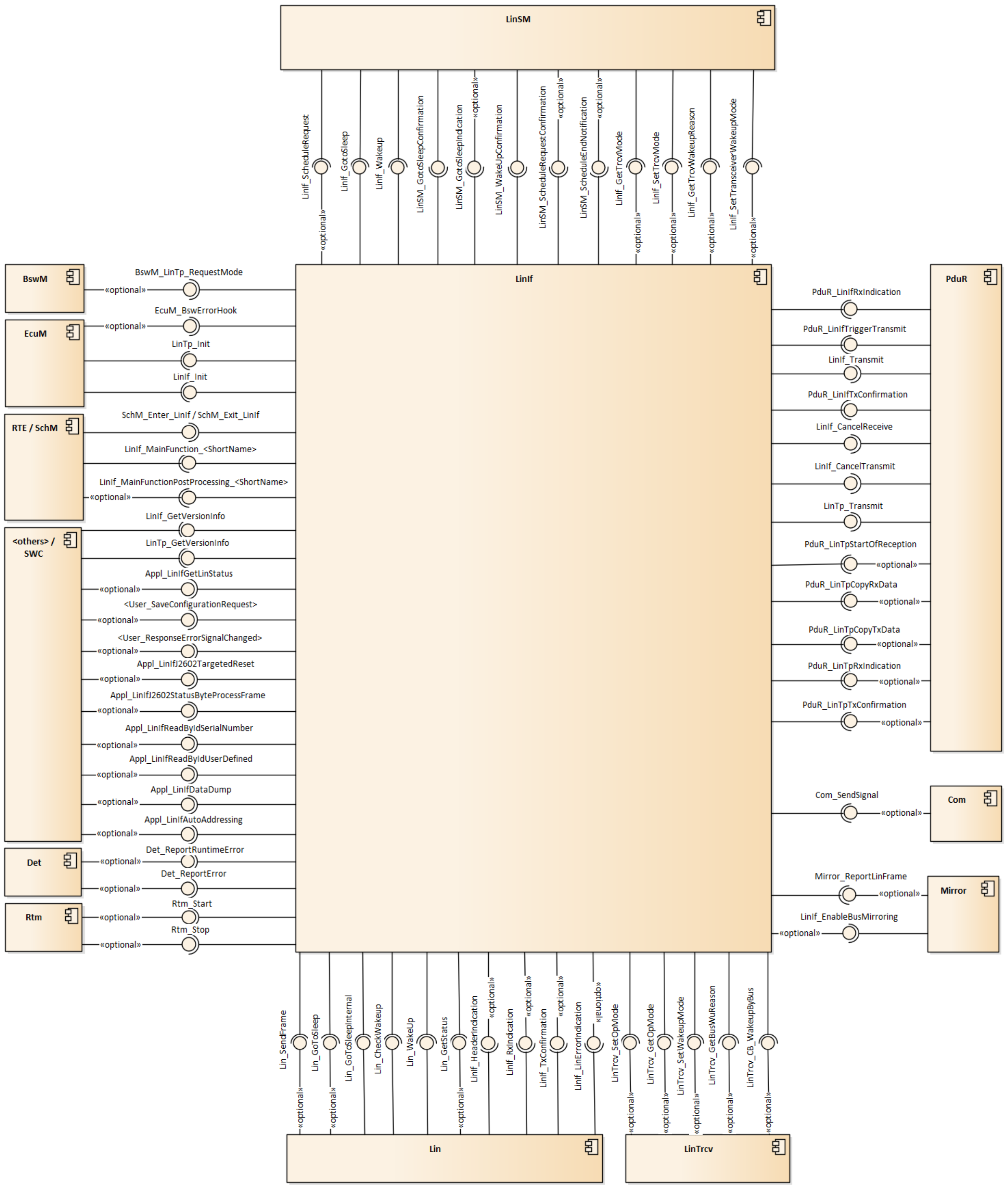


Figure 1-2 Interfaces to adjacent modules of the LINIF

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 does not provide any service ports. Optional callouts are directly called into the application layer without usage of port interfaces.

2 Functional Description

2.1 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 2-1 Supported AUTOSAR standard conform features

For information of not supported features see also chapter 6.

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

- Table 2-2 Features provided beyond the AUTOSAR standard

The following features specified in [1] are supported:

Supported AUTOSAR Standard Conform Features
Realization of LIN master and LIN slave nodes
LIN Interface initialization
(Multiple) LIN Driver handling
LIN Schedule table handling
LIN Interface main function handling
(Multiple) LIN Transceiver Driver support
LIN Network Management: Modes changes between SLEEP / OPERATIONAL (with notification)
Bus mirroring (as introduced in AUTOSAR 4.4)
Development error detection and reporting to DET
Multi-channel support
Version reporting
Event-Triggert-Frames (LIN master)
Sporadic Frames
Transport Protocol Layer
Diagnostic Frame Dispatcher (LIN slave)
Configuration variants
Node configuration and identification
Schedule table support with run once and run continuous
Post-Build Loadable
External wakeup support according to AUTOSAR 4.1.x
MICROSAR Classic Identity Manager using Post-Build Selectable
Channel based main functions

Table 2-1 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 (LIN master)
Handling of Response Pending Frames without PduR interaction (LIN master)
Optional handling of diagnostic functional requests (LIN master)
Automatic precompile optimizations
Explicit RAM initialization
Runtime Measurement Support
Extended handling of diagnostic broadcast requests (LIN master)
Schedule End notification (LIN master)
Schedule Info API (LIN master)
LIN driver status callout (LIN master)
J2602 header and response tolerance (LIN master)
Node Configuration Service: LIN 2.0 AssignFrameId (LIN slave)
Node Configuration Service: ReadByIdentifier with identifiers 1, 3, [16, 31] & [32-63] (LIN slave)
Node Configuration Service: DataDump (LIN slave)
Node Configuration Service: AutoAddressing (LIN slave)
Configurable LIN product identification (LIN slave)
TP connections with current configurable NAD (LIN slave)
Dedicated bus idle timeout duration after wakeup request (LIN slave)
SAE J2602 Targeted Reset node configuration service (LIN slave)
SAE J2602 Status Byte Handling (LIN slave)
Meta data for LIN Transport Protocol
Post processing of channel main function

Table 2-2 Features provided beyond the AUTOSAR standard

2.2 Initialization

2.2.1 Memory Initialization

AUTOSAR expects the startup code to automatically initialize the whole RAM content. However, not every startup code of embedded targets initializes the variables with the specific values as expected by LINIF. To avoid variables which may not be initialized as expected, the MICROSAR Classic LIN Interface provides an additional function `LinIf_InitMemory()` (see chapter 4.2.2) to initialize all module variables.

Refer also to chapter 4.2.2 '`LinIf_InitMemory`'.

2.2.2 Initialization procedure

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

The function `LinIf_InitMemory()` (see 2.2) initializes all variables of the LIN Interface.

The function `LinIf_Init()` initializes the channel independent and channel dependent states. It has to be called before any other service function of the LINIF, expect for `LinIf_InitMemory()`.

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 3.6.1 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.

Example
The required initialization sequence is:

```
LinIf_InitMemory();
LinIf_Init();
LinTp_Init();
```

2.3 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 via parameter 'LinIfStartupState', the transition to CHANNEL_SLEEP or CHANNEL_OPERATIONAL is performed during channel initialization according to Figure 2-1.

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 2.6 'Wake up Handling' for further details.

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 2.7 'Sleep Handling' for further details.

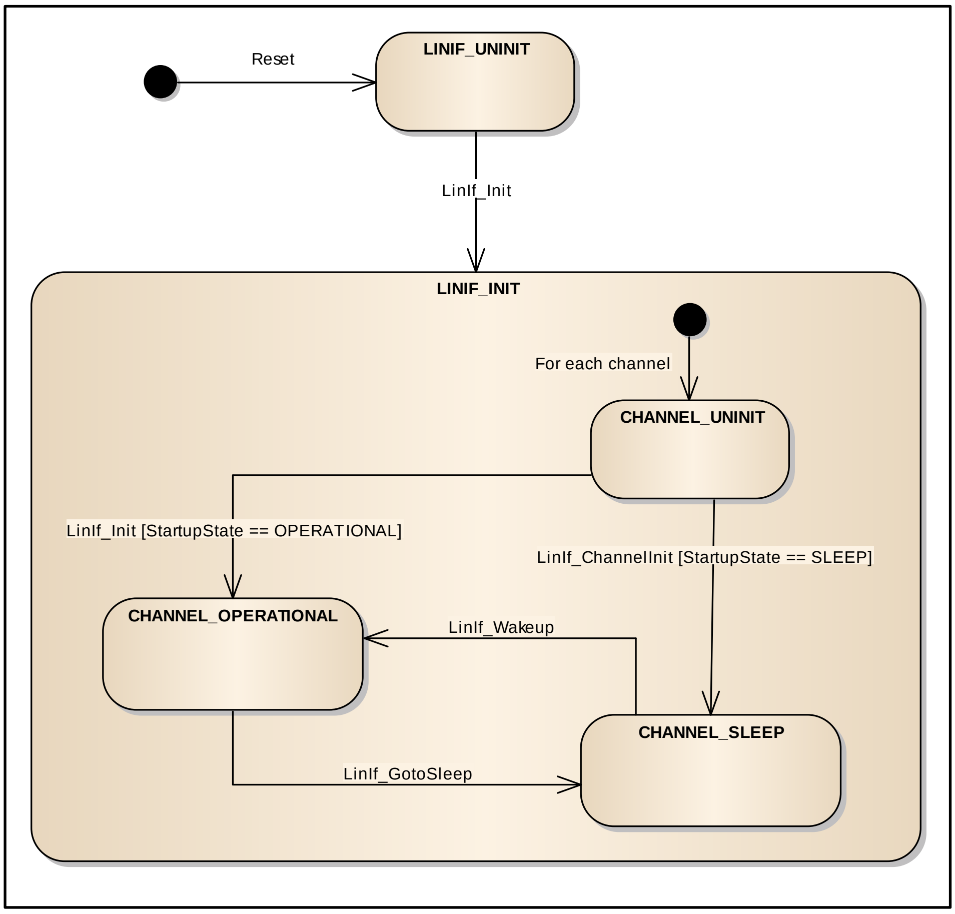


Figure 2-1 LIN Interface state machine

Also the startup state of a channel inside the LIN driver must be configured to support different initial states and AUTOSAR versions. If the initial channel states differ between LinIf and LIN driver, the LinIf ensures correct behavior by setting the LIN driver into the appropriate state using Lin_WakeupInternal() or Lin_GotoSleepInternal(). See also chapter 3.6.4 for details.

2.4 LIN Transport Protocol States

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

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 following states:

- LINTP_CHANNEL_IDLE
- LINTP_CHANNEL_BUSY

In case of 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 diagnostic frame was received, 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.

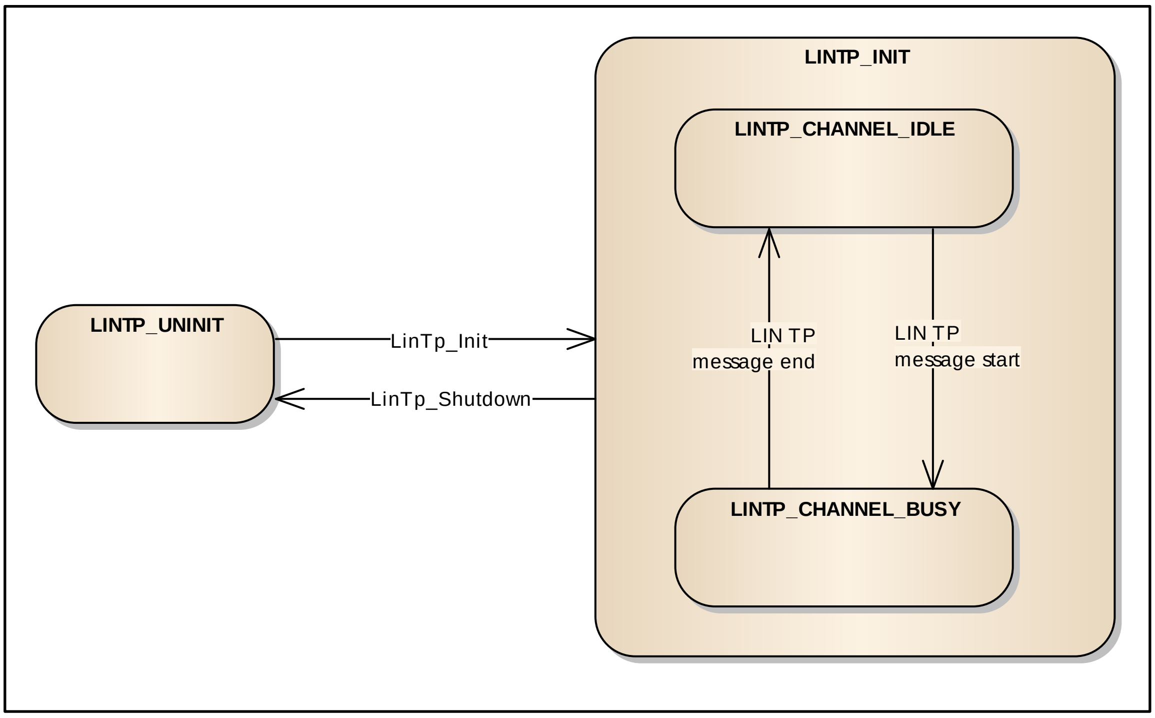


Figure 2-2 LinTp State machine

2.5 Main Functions

The LIN interface provides `LinIf_MainFunction_<LinIfChannel.ShortName>()` as a main function for each LinIf channel which performs following tasks:

- Schedule table handling for all channels (LIN master)

- Frame handling for all channels including transport protocol handling (LIN master)
- Timeout observations

Each channel has its own channel time base defined by parameter 'LinIfClusterTimeBase' which is by default the time base of the initial underlying database.

It must be ensured that `LinIf_MainFunction_<LinIfChannel.ShortName>()` is called with the configured cluster cycle time and minimal jitter to guarantee correct frame handling.

For LIN master nodes, the schedule and frame handling is processed inside the channel main function. All notifications and interactions to upper layer are delayed until the very end of the main function, called 'main function post processing' in the following. This design ensures a reduced execution duration within an exlusive area.

For special use cases, a highly synchronized execution of the main function of all channels might be required, e.g. to minimize the offset of the start of a schedule slot over several channels. The main function post processing of each LIN channel can be moved to its own main function using configuration parameter 'LinIfSeparateMainFunctionPostProcessing'. If enabled, then two main functions for each LIN channel are provided that can be scheduled by the OS / environment separately:

`LinIf_MainFunction_<LinIfChannel.ShortName>()`

`LinIf_MainFunctionPostProcessing_<LinIfChannel.ShortName>()`

Following integration requirement must be fulfilled for correct behavior:

- Each of the two main functions must be called with the configured cycle time for this LinIf channel ('LinIfClusterTimeBase' parameter)
- Both main functions of the same LinIf channel must be invoked alternatively, for each execution of `LinIf_MainFunction_<LinIfChannel.ShortName>()`, a call to `LinIf_MainFunctionPostProcessing_<LinIfChannel.ShortName>()` must follow before `LinIf_MainFunction_<LinIfChannel.ShortName>()` is executed again.
- Both main functions must not interrupt each other.

2.6 Wake up Handling

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

2.6.1 Internal wakeup

An internal wakeup sequences describes the transition from CHANNEL_SLEEP state to CHANNEL_OPERATIONAL state when no LIN wakeup frame was previously detected on the current channel. The handling depends on the implemented node type and current channel state:

- **LIN master nodes**:
  - 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 following behavior depends on the configuration (see chapter 3.6.4 for details):
  - Either the successful wakeup frame transmission is checked from the LIN driver after the maximum wakeup frame transmission duration is elapsed and afterwards the successful transition to state CHANNEL_OPERATIONAL is confirmed to upper layer by calling `<User>_WakeupConfirmation()`. This is the default case if a MICROSAR LIN driver is used.
  - Or the transition to state CHANNEL_OPERATIONAL is directly confirmed to upper layer by calling `<User>_WakeupConfirmation()` without a further check of the LIN driver status. This is the default case if a 3rd party AUTOSAR LIN driver is used.
- After the wakeup frame 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.

- **LIN slave nodes**:
  - 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 wakeup frame transmission itself is not confirmed by the LIN driver. If the LIN bus was successfully woken up, the LIN master will start scheduling LIN header / frames. With the reception of the first LIN header, the successful transition to CHANNEL_OPERATIONAL state is confirmed to upper layer by calling `<User>_WakeupConfirmation()`.

Expert Knowledge
The transition to CHANNEL_OPERATIONAL state is not performed inside `LinIf_Wakeup()` as defined by AUTOSAR, but postponed until the wakeup confirmation occurs. Otherwise the wakeup repetition mechanism of LinSM would not work correctly.

- The LinSM waits for a wakeup confirmation. In case the LIN master does not start scheduling (so the bus wakeup was not successful), a timeout occurs in LINSM. In this case, a retransmission of a wakeup can be configured in LINSM. The relevant parameters are 'LinSMConfirmationTimeout', 'LinSMSilenceAfterWakeupTimeout' and 'LinSMModeRequestRepetitionMax', see [8] for further details.
- The duration from wakeup request to wakeup confirmation is supervised by the bus idle timeout. The dedicated parameter 'LinIfWakeupConfirmationTimeoutPeriod' is used to allow the configuration of this wakeup confirmation timeout independently from the general bus idle timeout duration. The wakeup confirmation timeout is started in `LinIf_Wakeup()` and stopped by the confirmation of the wakeup. If the timeout elapses before the wakeup confirmation is detected, the sleep transition as described in 2.7.2 applies.

- CHANNEL_OPERATIONAL: If `LinIf_Wakeup()` is called in this state, nothing will be done and a successful wakeup is indicated to upper layer.

2.6.2 External wakeup

An external wakeup on the LIN bus is detected either by the LIN driver or the LIN transceiver (if available). An external wakeup can only be detected in CHANNEL_SLEEP state. 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.
- If afterwards the ComM user requests FULL COMMUNICATION, `LinIf_Wakeup()` is called. As the LINIF was informed about the preceeding external wakeup frame on the bus, no further wakeup frame is transmitted but the LIN driver is set to operational state, a successful wakeup is indicated to upper layer and CHANNEL_OPERATIONAL state is entered.

2.7 Sleep Handling

2.7.1 Sleep transition in LIN master nodes

A LIN master node sends a go-to-sleep frame to set the LIN bus into sleep mode.

Sleep mode handling is only applicable in state CHANNEL_OPERATIONAL. A go-to-sleep 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 go-to-sleep frame is scheduled (instead of the actual header specified in the schedule table). After the transmission of the go-to-sleep frame, the state CHANNEL_SLEEP is reached and the upper layer is informed by calling `<User>_GotoSleepConfirmation()`.

If the function `LinIf_GoToSleep()` is called in state CHANNEL_SLEEP, an internal sleep mode transition is executed. The function LinIf_GotoSleepInternal() is called on main function level to set the LIN driver into sleep state. Afterwards the upper layer is informed by calling `<User>_GotoSleepConfirmation()`.

2.7.2 Sleep transition in LIN slave nodes

A LIN slave node cannot actively set the LIN bus into sleep mode. It enters sleep mode caused by one of the following events:

- Reception of a go-to-sleep command transmitted by the LIN master.
- Detection of bus inactivity: No communication events are indicated by the LIN driver for the duration configured in parameter ‘LinIfBusIdleTimeoutPeriod’.

In either case, the upper layer is informed about the sleep event by calling `<User>_GotoSleepIndication()`. Afterwards, the actual transition to state CHANNEL_SLEEP must be requested by upper layer by calling the function `LinIf_GoToSleep()` which sets also the LIN driver into sleep state by calling `Lin_GotoSleepInternal()`. Finally the transition to CHANNEL_SLEEP state is confirmed to upper layer by calling `<User>_GotoSleepConfirmation()`.

2.8 Frame Handling

2.8.1 Frame Handling in LIN master nodes

LIN master nodes control the communication in a LIN network. The slots of an activated schedule table processed one after another. If a new schedule slot is due, the LINIF requests the LIN driver to either send a complete frame (Tx frame) or the header of frame (Rx frame or irrelevant frame) by calling `Lin_SendFrame()`. Before the next schedule slot is processed, the LINIF requests the LIN driver about the status of the previous frame / header transmission / reception and possible received data by calling `Lin_GetStatus()`.

All frame processing is performed on task level in the context of `LinIf_MainFunction_<LinIfChannel.ShortName>()`.

Following frame types are supported by LIN master nodes:

- Unconditional frames (Rx, Tx, Irrelevant)
- Event-triggered frames
- Sporadic frames
- Node configuration frames as fixed schedule table entries
- Diagnostic frames (over LINTP)

2.8.2 Frame Handling in LIN slave nodes

LIN slave nodes react on the reception of LIN header that are transmitted by the LIN master node and are indicated by the LIN driver with the callback `LinIf_HeaderIndication()`. The received PID is evaluated to distinguish if the response shall be transmitted, received or ignored. A received response, including the received data, is indicated by the LIN driver with the callback `LinIf_RxIndication()`. A response transmission is confirmed with callback `LinIf_TxConfirmation()`. Any communication error that is detected by the LIN driver is reported with callback `LinIf_LinErrorIndication()`. All frame processing is performed on interrupt level in the context of the LIN driver callback functions.

Following frame types are supported by LIN slave nodes:

- Unconditional frames (Rx, Tx, Irrelevant)
- Sporadic frames (reception of associated unconditional frames)
- Diagnostic frames (Node configuration services and LINTP)

2.9 Schedule Table Handling (LIN master)

Note
This chapter is only applicable to LIN master nodes.

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 [4].

2.9.1 Schedule Table Handling after Wakeup

According to [5], 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 Classic 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 LinIfClusterTimeBase parameter.

2.10 Diagnostic Communication

Diagnostic communication in LINIF can be distinguished between:

- LIN Node Configuration and Identification services (see 2.12 for further details)
- Diagnostics over LIN Transport Protocol (see 2.11 for further details)

Diagnostic communication data is always transferred in predefined frames on LIN:

Diagnostic requests are transmitted by the LIN master node in Master Request Frames (MRF) with ID 0x3C and received by LIN slave nodes.

Diagnostic responses are transmitted by LIN slave nodes in Slave Response Frames (SRF) with ID 0x3D and received by the LIN master nodes.

MRF and SRF do implicitly always exist on the LINIF, even if they might not be configured.

FAQ
Master Request and Slave Response Frames are either used internally for node configuration services or are accessed by LIN Transport Protocol. There is no direct interface to access MRF / SRF from application layer using solely LINIF.

2.10.1 Diagnostic Communication Dispatching

Because the MRF and SRF are shared between gateway / application diagnostics (LINTP) and Node Configuration Services, the LINIF 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.

LIN master nodes:

An ongoing LinTp connection is aborted if a schedule table slot with a LIN node configuration service is activated. An scheduled SRF slot is processed solely by the current user, depending if the preceeding transmitted MRF was a node configuration reqeust or a LINTP diagnostic request.

LIN slave nodes:

A received MRF is evaluated if it contains a go-to-sleep frame, a node configuration request or a diagnostic request.

An ongoing LinTp connection is aborted if a node configuration request is received addressing the LIN slave node. A subsequent received SRF header is processed solely by the current user, depending if the preceeding received MRF was a node configuration reqeust or a LINTP diagnostic request.

2.11 LIN Transport Protocol

The LIN TP is used to transport diagnostic requests and responses. No diagnostic sessions are made in parallel. All service requests are always proceeded in sequence, with exception of functional requests (refer to 2.11.1 for further details).

For LinTp connections the two dedicated frames MRF and SRF 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 MRF (LIN master) respectively SRF (LIN slave) for transmission. Analogous, for reception of a diagnostic response an RxNsdu has to be configured and all RxNSdus on a channel share the SRF (LIN master) respectively MRF (LIN slave) for reception.

In general, the NAD to be used is statically configured for each TxNsdu and RxNSdu. Generic connections can be configured by using meta data which allow connections that are independent of the configured NAD value, refer to 2.11.7 for further details.

Only one (physical) LinTp connection can be active at a certain time. For LIN master nodes, an ongoing LinTp reception is aborted inside a `LinTp_Transmit()` call and the new diagnostic request is **accepted.** For LIN slave nodes, an ongoing LinTp reception is continued when `LinTp_Transmit()` is called and the new request is **ignored**.

LIN master nodes:

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.

A diagnostic transmission, requested by `LinTp_Transmit()`, uses the NAD that is configured for the used TxNSdu in case of a static connection or uses the provided NAD from upper layer in case of a generic connection.

LIN slave nodes:

A diagnostic reception is by default only accepted if an RxNSdu is configured with a NAD matching the received NAD.

As the configured NAD of a LIN slave node may change during runtime, it is possible to enable 'LinTpRxNSduConfigurableNADSupported' for one RxNSdu of each LIN slave channel. In this case, the received NAD is additionally compared to the currently configured NAD and if it matches, the reception is accepted and this RxNSdu is used.

A diagnostic transmission, requested by `LinTp_Transmit()`, uses the NAD that is configured for the used TxNSdu. As the configured NAD of a LIN slave node may change during runtime, it is possible to enable 'LinTpTxNSduConfigurableNADSupported' for one TxNSdu of each LIN slave channel. In this case, the currently configured NAD is used for transmission of this TxNSdu. For a generic connection, the NAD provided by upper layer is used for transmission.

2.11.1 Functional Requests

A functional request (NAD = 0x7E) may interleave the physical diagnostic communication at any time. The handling of functional requests can be disabled if not required by parameter 'LinTpFunctionalRequestsSupported' for optimization reasons.

LIN master nodes:

The functional request is transmitted always first when pending and no response is expected.

LIN slave nodes:

A functional request is received and forwarded to upper layer if no other LINTP connection is active on the channel. Otherwise the functional request is ignored.

2.11.2 Response Pending Frames

LIN master nodes:

Extended timeout observation based on parameters 'LinTpP2Timing' and 'LinTpP2Max' (refer also to chapter 2.11.5) 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 2-3 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.

LIN slave nodes:

No special handling is required for response pending frames. The transmission is requested via `LinTp_Transmit`() and is processed as any other LINTP single frame.

2.11.3 Broadcast Requests

LIN master nodes:

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.

LIN slave nodes:

A broadcast request from the LIN master is received by a LIN slave node if an RxNSdu with NAD = 0x7F is configured. It is up to the upper layer to not request the transmission of a response in this case to avoid collisions.

2.11.4 Diagnostic Schedule Table Change (LIN master)

Note
This chapter is only applicable to LIN master nodes.

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 [4].

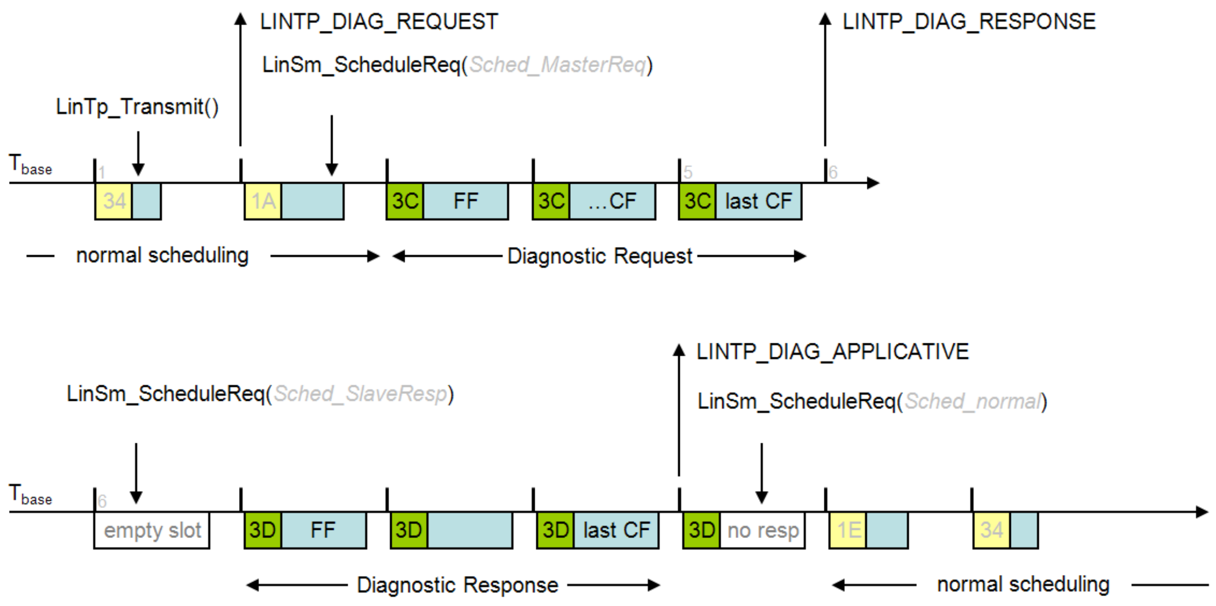


Figure 2-3 Diagnostic communication and schedule table changes

2.11.5 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 2.11.4) is not correctly implemented or no MRF / SRF is scheduled (LIN master) or received (LIN slave). For LIN master nodes, 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 (LIN master)

Transport layer timing parameters are configured per Tx/Rx connection and are defined in Table 2-4:

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 2-4 Transport Layer timing constraints

Diagnostic timing parameters are only applicable to LIN master nodes. They are globally configured for the LinTp and defined in Table 2-5:

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 2-5 Diagnostic timing constraints (LIN master)

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

Following figures visualizes the different phases with active timeout observation:

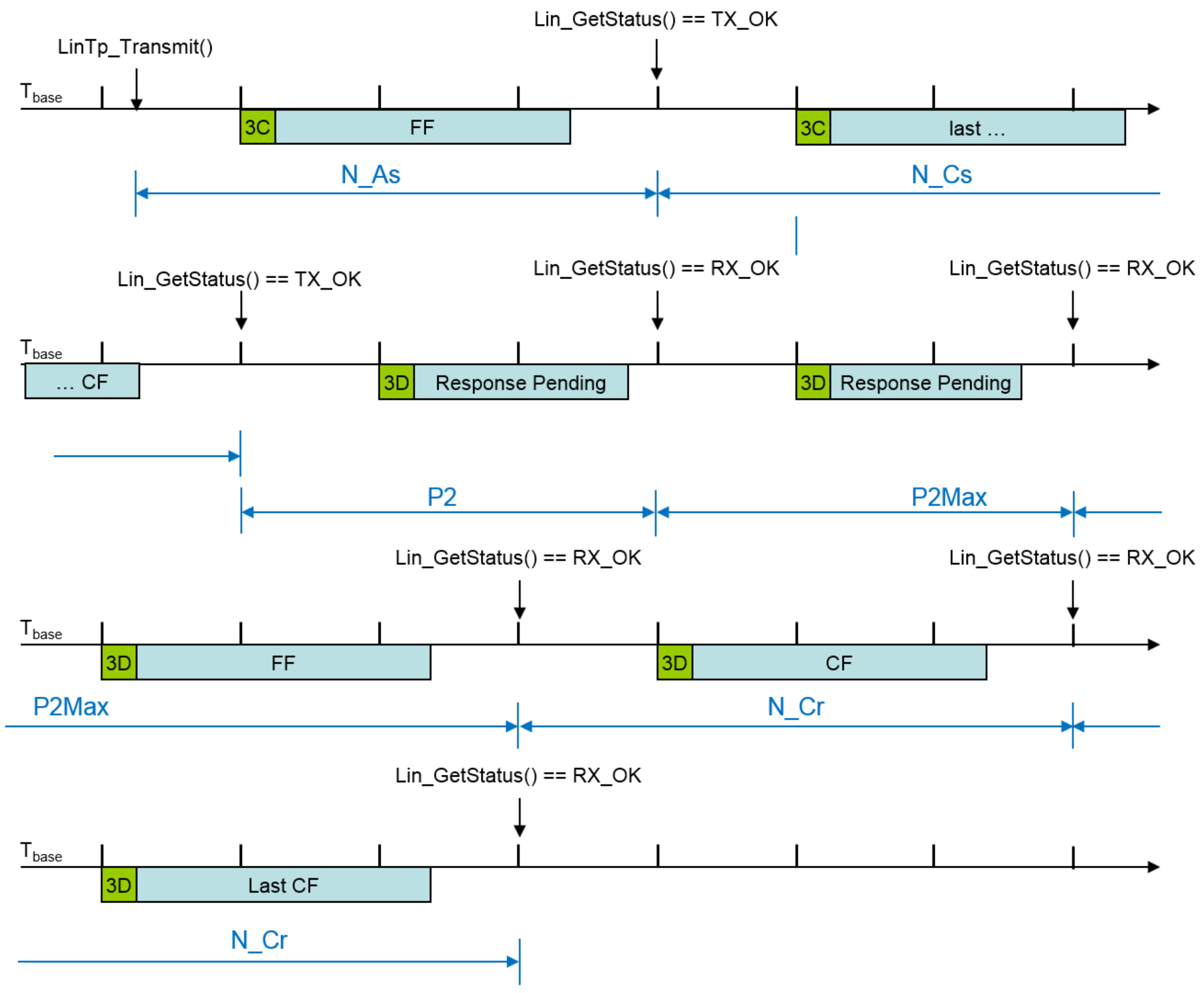


Figure 2-4 LinTp Timeouts (LIN master)

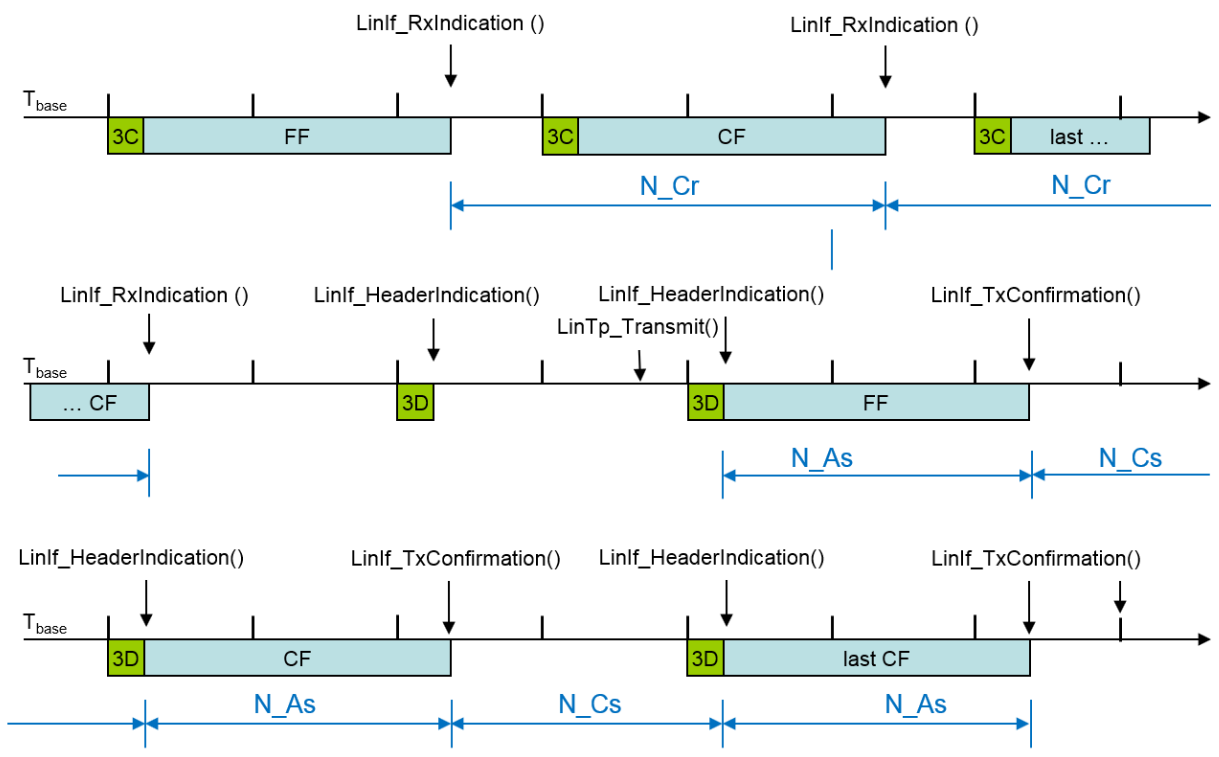


Figure 2-5 LinTp Timeouts (LIN slave)

2.11.6 Application-specifc diagnostics communication

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 [14])
- 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 and to get notified about the payload bytes of the diagnostic reception.
- Meta data can be used to provide and receive the used NAD at runtime instead of statically configure it in each NSDU container, see 2.11.7 for further details.

Figure 2-6 shows an example case with one LIN bus where the Tx and Rx routing paths between Dcm and LinTp to slave node with NAD 0x01 are preconfigured or derived. The support of additional application-specific diagnostics of slave node with NAD 0x01 can be added by duplicating the routing paths and changing the upper layer module to CDD.

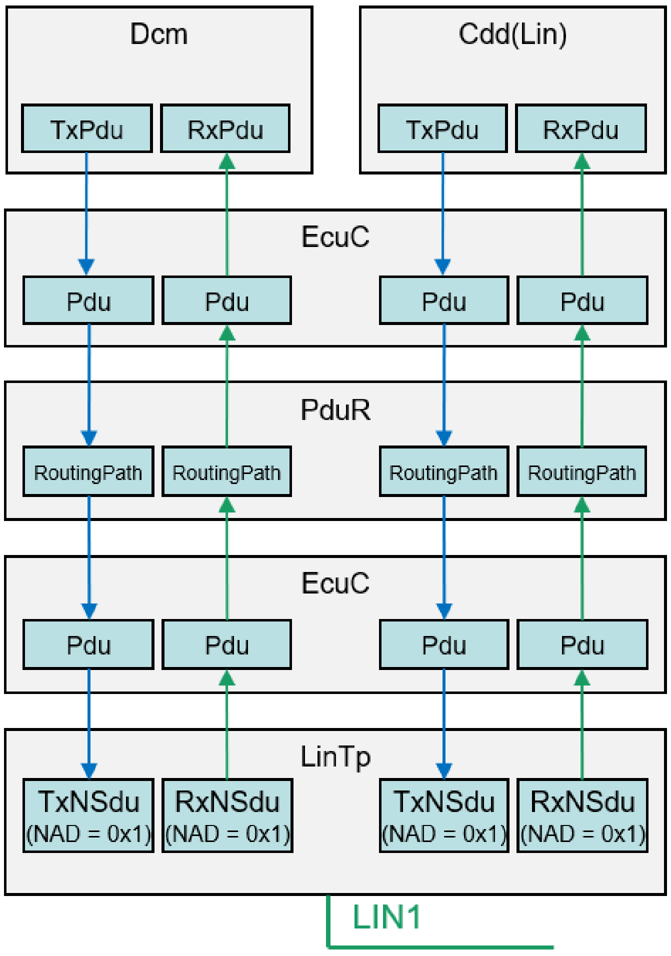


Figure 2-6 Diagnostic communication configuration example

2.11.7 Generic connections using meta data

Meta data handling allows the usage of so-called generic connections for which the addressing information is determined dynamically at runtime. This means that the NAD for a TxNSdu or RxNSdu is not statically defined in the configuration. Instead, for diagnostic transmission the NAD is provided by the upper layer during runtime and for diagnostic reception the LinTp passes the received NAD to upper layer.

The usage of meta data is configured for each routing path separately. To enable meta data for a routing path, create the meta data length parameter for the referenced PDU (`EcuC/EcucPduCollection/Pdu/MetaDataLength`) in the ECUC module and set it to 1. The meta data support is implemented according to AUTOSAR 4.2.2.

LIN master nodes:

It is required that for each TxNSdu using meta data, there is a corresponding RxNSdu which also uses meta data on the same LIN channel referring to the same upper layer module of the routing path.

LIN slave nodes:

If meta data is enabled as well as the configurable NAD option is used for a TxNSdu, then the configurable NAD option has higher priority so that the current configured NAD is used for transmission and the NAD provided as meta data is ignored.

2.12 Node configuration and identification services

2.12.1 Node configuration services in LIN master nodes

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. See 2.11.6 for further details.

2.12.2 Node configuration services in LIN slave nodes

Each received MRF is evaluated if it contains a node configuration service request with a configurered SID. In this case, the service is processed by LINIF as described below and the request is not further forwarded to the LINTP. Otherwise the frames are provided to LINTP.

Each node configuration request is checked for validitiy, i.e. for correct NAD, PCI and service-specific data. Wildcards are supported for Supplier and Function ID.

If a node configuration request is accepted, a response is set pending and a N_As timeout is started (parameter 'LinIfNasTimeout'). The pending response is transmitted with the next received SRF header. If the timeout elapses before an SRF header is received, the response is discarded.

2.12.2.1 Supported services

2.12.2.1.1 The AssignNAD

The AssignNAD service is supported if the LIN protocol version of the slave is greater or equal to LIN 2.0 and the parameter 'LinIfNcOptionalRequestSupported' is enabled.

If the request is accepted, the configured NAD is updated with the provided NAD and a positive response is transmitted. No interaction with upper layer or application occurs. An invalid request is ignored without further action (no response is transmitted).

2.12.2.1.2 AssignFrameIdentifierRange

The AssignFrameIdentifierRange service is supported if the LIN protocol version of the slave is greater or equal to LIN 2.1.

If the request is accepted, the PIDs of the requested frame are updated with the provided values and a positive response is transmitted. The “unassign PID” and “Do-no-care PID” values are supported. No interaction with upper layer or application occurs. An invalid request is ignored without further action (no response is transmitted).

2.12.2.1.3 ReadByIdentifier

The ReadbyIdentifier service is supported according to following table:

Identifier	Supported if…
0 (LIN product identification)	the LIN protocol version of the slave is greater or equal to LIN 2.0
1 (Serial number)	the LIN protocol version of the slave is greater or equal to LIN 2.0 and ReadByIdentifier support for identifier 1 is enabled (parameter ‘LinIfReadByIdentifierSerialNumberSupported’)
2 (Bit timing test)	the LIN protocol version of the slave is greater or equal to ISO 17987
3 (Database version)	the LIN protocol version of the slave is greater or equal to ISO 17987 and data base version support is enabled (parameter ‘LinIfReadByIdDatabaseVersionSupported’)
[15, 31] (LIN 2.0 message ID)	AssignFrameId (LIN 2.0) service support is enabled (see 2.12.2.1.5.)
[32, 63] (User defined ID)	the LIN protocol version of the slave is greater or equal to LIN 2.0 and ReadByIdentifier support for identifier [32, 63] (parameter ‘LinIfReadByIdentifierUserDefinedSupported’) is enabled
other	Not supported

Table 2-6 ReadByIdentifier support

If the request is supported and accepted, the subsequent handling depends on the received identifier:

- Identifier 0, 3 and [15, 31]: A positive response is transmitted.
- Identifier 1: The callout function `Appl_LinIfReadByIdSerialNumber` (see 4.6.4 for further details) is executed. The application decides the type of the response (positive or negative response) by the return value and provides the serial number in case of a positive response.
- Identifier [32, 63]: The callout function `Appl_LinIfReadByIdUserDefined` (see 4.6.5 for further details) is executed. The application decides the type of the response (positive, negative or no response) by the return value and provides the return data and return length in case of a positive response.
- Identifier 2 and any other / unsupported identifiers: A negative response is transmitted.

2.12.2.1.4 SaveConfiguration

The SaveConfiguration service is supported if the LIN protocol version of the slave is greater or equal to LIN 2.1 and the SaveConfiguration callout is configured (parameter 'LinIfSaveConfigurationCallout').

If the request is accepted, the configured SaveConfiguration callout function is executed. Depending on the return value, either a positive response or no response is transmitted.

2.12.2.1.5 AssignFrameId (LIN 2.0)

The AssignFrameId (LIN 2.0) service is supported if the LIN protocol version of the slave is equal to LIN 2.0 and the parameter 'LinIfNcOptionalRequestSupported' is enabled.

If the request is accepted, the PID of the frame identified by the provided message ID is updated with the provided PID value and a positive response is transmitted. No interaction with upper layer or application occurs. An invalid request is ignored without further action (no response is transmitted).

2.12.2.1.6 Targeted Reset (SAE J2602)

The TargetedReset service is supported if the LIN protocol version of the slave is set to SAE J2602.

If a valid request is received, the TargetedReset callout function `Appl_LinIfJ2602TargetedReset()` (see 4.6.2) is executed. The application decides the type of the response (positive response, negative response or no response) by the return value. The NAD of the request is provided to distinguish physical and broadcast requests. In case of a broadcast reset request, no response is sent at all, independent of the provided return value. If a positive response is requested, the reset state in the J2602 status byte is set.

2.12.2.1.7 DataDump

The DataDump service is supported if the LIN protocol version of the slave is equal to or greater than LIN 2.0 and the parameter 'LinIfDataDumpSupported' is enabled.

If a valid request is received, the DataDump callout function `Appl_LinIfDataDump()` (see 4.6.6) is executed. The application decides the type of the response (positive response or no response) by the return value. In case of a positive response, the application provides the response data.

2.12.2.1.8 AutoAddressing

The AutoAddressing service is supported if the LIN protocol version of the slave is equal to or greater than LIN 2.0 and the parameter 'LinIfAutoAddressingSupported' is enabled.

If a valid request is received, the AutoAddressing callout function `Appl_LinIfAutoAddressing()` (see 4.6.7) is executed. The application decides the type of the response (positive response or no response) by the return value.

The default SID of the AutoAddressing service is 0xB8. In previous LIN specifications, the AutoAddressing service has been called SNPD (Slave Node Position Detection) and has used SID 0xB5. It is possible to use the legacy SID via a user config file with following content:

```
#define LINIF_NC_AUTO_ADDRESSING_SID_B5  STD_ON
```

Due to the collision with the SID for SAE J2602 Targeted Reset service, this redefinition of the SID is not supported in combination with SAE J2602 protocol version.

2.12.2.2 LIN Product Identification

Each slave node provides a Supplier ID, Function ID and Variant ID. This information is statically provided in the configuration. In order to change the slave product identification during runtime, enable the option 'LinIfProductIdentificationConfigurableSupported' to active the API function `LinIf_SetLinProductIdentification()`.

2.13 Error Handling

2.13.1 Development Error Reporting

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

If another module is used for development error reporting, the function prototypes for reporting the error can be configured by the integrator, but must have the same signatures as the services `Det_ReportError()` and `Det_ReportRuntimeError()`.

The reported LINIF ID is 62.

The reported service IDs identify the services which are described in 4.2. 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`
0x0E	`LinIf_SetLinProductIdentification`
0x0F	`LinIf_SetJ2602StatusByteResetBit`
0x40	`LinTp_Init`
0x41	`LinTp_Transmit`

Service ID	Service
0x42	`LinTp_GetVersionInfo`
0x43	`LinTp_Shutdown`
0x44	`LinTp_ChangeParameter`
0x46	`LinTp_CancelTransmit`
0x47	`LinTp_CancelReceive`
0x60	`LinIf_CheckWakeup`
0x61	`LinIf_WakeupConfirmation`
0x70	`LinIf_GetConfiguredNAD`
0x71	`LinIf_SetConfiguredNAD`
0x72	`LinIf_GetPIDTable`
0x73	`LinIf_SetPIDTable`
0x78	`LinIf_HeaderIndication`
0x79	`LinIf_RxIndication`
0x7A	`LinIf_TxConfirmation`
0x7B	`LinIf_LinErrorIndication`
0x7F	`LinIf_EnableBusMirroring`
0x80	`LinIf_MainFunction_<LinIfChannel.ShortName>`

Table 2-7 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.
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.
0x71	LINIF_E_CONFIG	Invalid configuration, because configuration structure not valid. Has been introduced by Vector Informatik GmbH.

Table 2-8 Errors reported to DET

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

Error Code		Description
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.
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 2-9 Runtime errors reported to DET

2.13.2 Production Code Error Reporting

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

2.14 LIN Communication Error Detection

During development, short-term communication problems are detected by evaluating the status value of the LIN driver (LIN master) respectively by evaluating the indicated error status (LIN slave) 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 2-10 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

Error Type	Detection by Com/SWC
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 2-10 Communication error types and detection

For further information about COM timeout monitoring, please refer to [13].

Expert Knowledge
In order to get the LIN driver status on frame level, the callout function `Appl_LinIfGetLinStatus()` could additionally be used. See 4.6.1 for further details.
This is only applicable to LIN master nodes.

2.14.1 Response Error Signal (LIN slave)

Note
This chapter is only applicable to LIN slave nodes.

Each LIN slave node with LIN protocol version greater or equal to LIN 2.0 provides a specific one-bit scalar signal. This response error signal provides the status of the LIN slave to the network.

The value of the response error signal is completely controlled by the LINIF. The application / SWC must not alter the signal value. Every time the LINIF detects a condition causing a change of the response error signal value, it calls `Com_SendSignal`() to update the signal value. The application may get notified about each change of the response error signal by configuring a callout function (‘LinIfResponseErrorSignalChangedCallout’ parameter).

The LINIF sets the response error signal if a communication error is indicated by the LIN driver for a frame response that is received or transmitted by the slave node.

The LINIF clears the response error signal when the unconditional frame containing the response error signal is successfully transmitted.

Expert Knowledge
The response error signal must be contained in exactly one unconditional Tx frame.

For LIN slave nodes using LIN protocol version SAE J2602(2012), the response error signal handling is replaced by the J2602 status byte handling, see 2.17 for further details.

2.15 Main Function jitter consideration (LIN master)

Note
This chapter is only applicable to LIN master nodes.

The `LinIf_MainFunction_<LinIfChannel.ShortName>()` is responsible to process the schedule table. In detail, if a new schedule slot is due, it starts the frame by calling the `Lin_SendFrame()` service of the underlying LIN driver and after the maximum frame duration has elapsed it retrieves the frame status from the LIN driver by calling `Lin_GetStatus()`.

This implies that any delay in `LinIf_MainFunction_<LinIfChannel.ShortName>()` invocation has a direct impact on the schedule slot timings on the LIN bus. If this main function jitter becomes too large (e.g. under high system load peeks), the slot timings could vary in such an extent that the frame of the following schedule slot is started while the current frame is still active. This causes the pending frame to be aborted due to the start of the LIN header of the next frame.

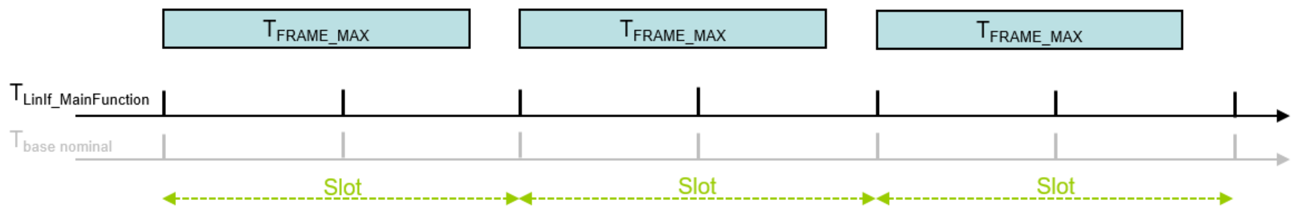


Figure 2-7 LinIf main function calls without jitter

Figure 2-7 shows the invocations of the LinIf main function without jitter corresponding to the LIN time base, resulting in perfect schedule table slot timings on the LIN bus. In Figure 2-8 below, the main function jitter causes a delay so that the previous schedule slot is stretched and shortens the current schedule slot by the same amount. If the actual frame duration is longer than the contracted schedule slot, the next schedule slot is already started while the current frame is still being processed and overlaps it.

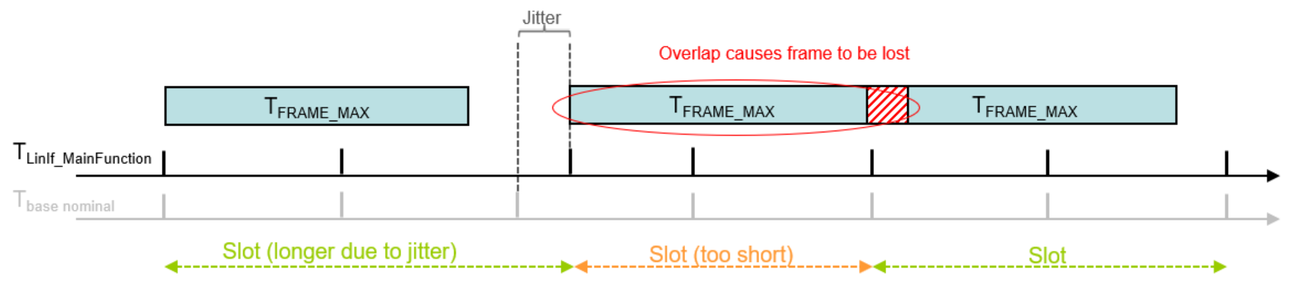


Figure 2-8 Delayed LinIf main function call caused by jitter

This effect causes the loss of frames and is observable by following indicators:

- Missing Rx indications or Tx confirmations from LinIf to upper layer (`Lin_GetStatus()` returns `LIN_RX_BUSY` or `LIN_TX_BUSY`)

- DET error LINIF_E_RESPONSE is reported (if enabled)
- Possible errors in the CANoe communication trace (e.g. incomplete frame, framing error while waiting for checksum, …)

The maximum duration of a LIN frame is defined as T_{FRAME_MAX} in [7]. To provide frame confirmations and indications with the latest reception data as fast as possible, the LinIf calls `Lin_GetStatus()` in the subsequent LinIf main function call after the T_{FRAME_MAX} has elapsed.
In case of sporadic communication problems which are caused by jitter of the main function invocations, perform following steps:

- Analyze the worst-case LinIf main function jitter in the system, e.g. by measurement.
- Configure the retrieved maximum jitter value in LinIf under: `LinIf/LinIfGlobalConfig/LinIfChannel/LinIfNodeType/LinIfMaster/LinIfJitter`. Note this might cause to overwrite a preconfigured value using the “user-defined” mechanism.
- This jitter value is considered during calculation of the `Lin_GetStatus()` call point in time. Enlarging the jitter value might delay the `Lin_GetStatus()` call inside a schedule slot.

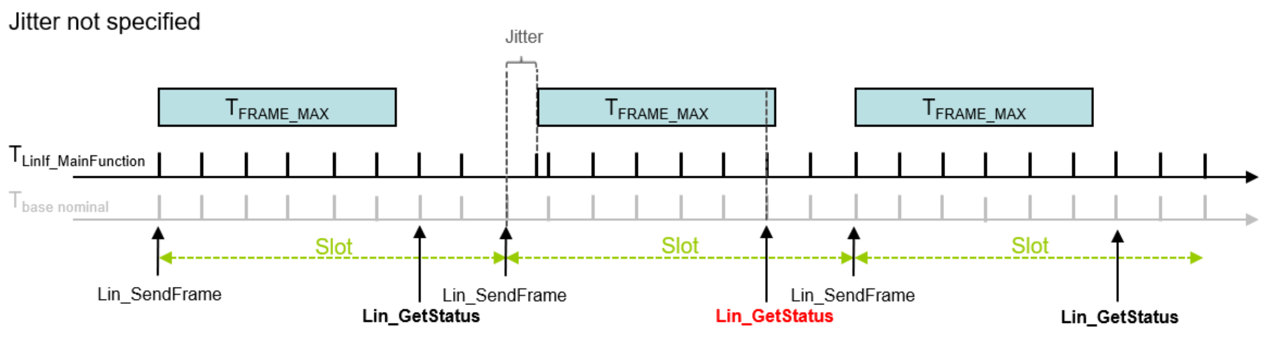


Figure 2-9 Calculated Lin_GetStatus() call timings without jitter

Above Figure 2-9 illustrates the `Lin_GetStatus()` call timings without any jitter specified: It is called in the following main function cycle after the maximum frame duration has elapsed. In case the main function is delayed, `Lin_GetStatus()` might be called while the frame is still being processed and the frame is lost (Function call marked in red).

If the actual jitter is specified, it is considered in the calculated `Lin_GetStatus()` call points in time as shown in following Figure 2-10. If a main function call is delayed by the specified jitter duration at maximum, the frame is still correctly processed because the `Lin_GetStatus()` function is called after the maximum frame duration plus the specified jitter.

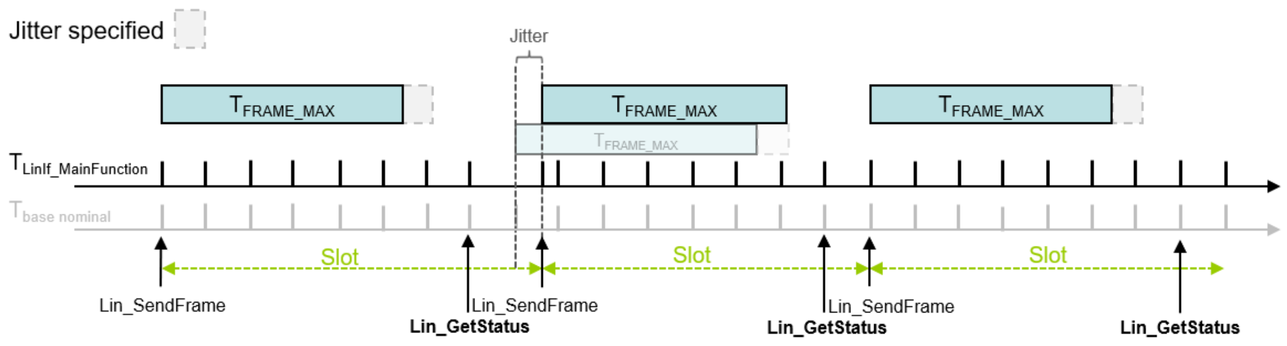


Figure 2-10 Calculated Lin_GetStatus() call timings with specified jitter considered

Caution
Increasing the LinIf jitter value also increases the calculated maximum duration to handle the frame inside a schedule slot. This could cause a validation warning in the configuration tool during generation if the calculated maximum frame time including jitter exceeds the schedule slot time.

Depending on the specific schedule timings, those validation warnings might be reduced by decreasing the channel call cycle time of the task function `LinIf_MainFunction_<LinIfChannel.ShortName>()`.

Note that solely enlarging schedule slot times without modifying the jitter value has no impact on the `Lin_GetStatus()` call point in time.

The calculated maximum frame duration including jitter is a worst-case upper bound limit, so the actual frame durations (even with jitter) on the bus could still be less causing no communication problems at all. It is up to the system designer to analyze all affected schedule slot timings and evaluate if the schedule slot delay shall be increased or if this specific slot is not affected (e.g. Tx frames might not be affected and always be transmitted within the nominal frame time in case the LIN driver uses a LIN hardware controller).

Reference
Please also consider 3.6.2 to minimize the jitter of the LinIf main function call in general.

2.16 SAE J2602 Frame Tolerance (LIN master)

Note
This chapter is only applicable to LIN master nodes.

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.

2.17 SAE J2602 Status Byte (LIN slave)

Note
This chapter is only applicable to LIN slave nodes.

For LIN slave nodes using SAE J2602 (2012) as LIN protocol version, LinIf provides the handling of the 3-bit error field of the J2602 status byte as described in [15]. It replaces the response error signal handling for LIN 2.x and ISO17987 slave nodes in this case.

A LIN communication error code reported from LIN driver is mapped to an appropriate J2602 error state and stored locally inside LinIf. Also the reset state of the J2602 status byte is supported. The LinIf calls `Com_SendSignal`() to update the 3-bit error field with the currently highest prior pending J2602 error state for all referenced unconditional transmit frames containing the status byte. Once an error state has been successfully transmitted, the error field is updated via `Com_SendSignal`() either with the next pending error state according to the priority or it is cleared if no further error state needs to be reported ("sticky bits").

Note
The application information field (APINFO4:0) of the J2602 status byte is not handled by LinIf. The setting of this field is task of the application.

The conditions to set the J2602 error states are listed in following table:

J2602 error state	Trigger condition
Reset	▶ A Targeted Reset request is received and accepted. ▶ `LinIf_SetJ2602StatusByteResetBit()` is called.
Data error	▶ `LinIf_LinErrorIndication()` is called with error code LIN_ERR_RESP_DATABIT for a relevant response.
Checksum error	▶ `LinIf_LinErrorIndication()` is called with error code LIN_ERR_RESP_CHKSUM for a relevant response.
Byte field framing error	▶ `LinIf_LinErrorIndication()` is called with error code LIN_ERR_RESP_STOPBIT for a relevant response.
ID parity error	▶ `LinIf_LinErrorIndication()` is called with error code LIN_ERR_HEADER.

Table 2-11 J2602 error state conditions

Caution
SAE J2602 expects the detection of invalid sync field values, framing errors in sync field and PID field as well as invalid parity bits in the PID field. These error types cannot be distinguished because the LIN driver only reports LIN_ERR_HEADER error code for ANY communication error during header reception according to AUTOSAR specification. For all these communication errors, ID parity error state is reported in the status byte.

This affects SAE J2602 (2012) conformity.

According to SAE J2602, the status byte shall be contained in every unconditional Tx frame. Depending on the signal modelling, more than one J2602 status error signal might exist in the Com module, so the LinIf supports the reference of one or several error status signals as response error signals in the configuration. If the LinIf detects an update of the J2602 error status, it calls `Com_SendSignal()` for each configured signal reference to ensure that the values of all error status signal instances are always synchronized.

For special use cases, one or several frames might not contain the SAE J2602 status byte. By default, the LinIf assumes the status byte to be contained in every unconditional Tx frame and updates the status byte (if required) after each frame transmission – potentially also after transmission of frames without status byte which would lead to loss of error reportings. The callout function `Appl_LinIfJ2602StatusByteProcessFrame()` can be used to support this use case. If enabled, the LinIf requests the application after each frame transmission if the current frame contains a J2602 status byte or not.

Caution
Unconditional Tx frames without status byte affect SAE J2602 (2012) conformity.

In line with the response error signal handling, the application may get notified about a change in the error state by configuring a callout function (see 4.5.2.1). It is called with parameter TRUE if an error state in the J2602 status byte is set while no error state has been active previously. It is called with parameter FALSE if an error state has been successfully transmitted and no more error state is pending (the error state will be cleared). The reset state is not treated as error state and excluded from this handling, so the callout is never called if the reset state is set or cleared.

3 Integration

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

3.1 Scope of Delivery

The delivery of the LINIF contains the files which are described in the chapters 3.1.1 and 3.1.2:

3.1.1 Static Files

File Name	Description	Integration Tasks
LinIf.h	Header containing the public interface of the LIN Interface.	-
LinIf.c	C code containing the general functionality of the LIN Interface.	-
LinIf_Master.h	Header containing the internal master specific interface of the LIN Interface.	-
LinIf_Master.c	C code containing the master specific functionality of the LIN Interface.	-
LinIf_Slave.h	Header containing the internal slave specific interface of the LIN Interface.	-
LinIf_Slave.c	C code containing the slave specific functionality of the LIN Interface.	-
LinIf_Int.h	Header containing the internal general interface of the LIN Interface.	-
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 3-1 Static files

3.1.2 Dynamic Files

The dynamic files are generated by the configuration tool.

File Name	Description	Integration Tasks
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).	-

File Name	Description	Integration Tasks
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 3-2 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	Integration Tasks
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 3-3 Generated files for LIN Transport Protocol

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

3.2 Include Structure

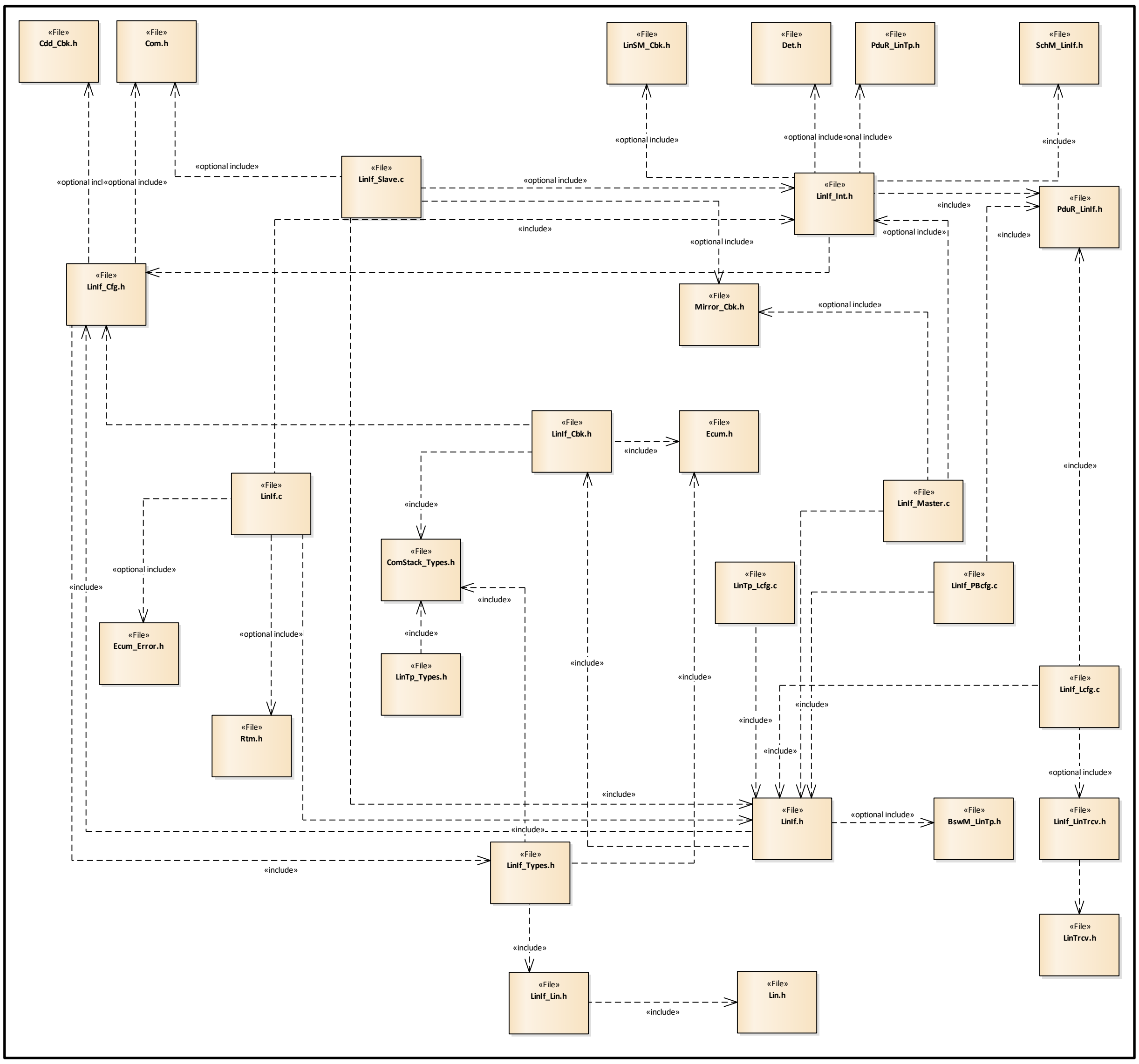


Figure 3-1 Include structure

Optional includes	Description
BswM_LinTp.h	This header file only included if LinTp support is enabled (`LINIF_TP_SUPPORTED == STD_ON`).
CDD_Cbk.h	Header files of CDD are only included if CDDs are used as configurable upper layer interfaces.
Com.h	This header file is included if LIN slave nodes are supported.
Det.h	Det.h is only included if reporting to DET is enabled (`LINIF_DEV_ERROR_REPORT == STD_ON`).
EcuM_Error.h	This header file is only included if the configuration variant Post-Build-Loadable or Post-Build-Selectable is active.
LinIf_LinTrcv.h	This header is only generated and included if the LIN Transceiver Driver Handling is enabled.
LinSM_Cbk.h	This header file is included if LinSM is used as configurable upper layer interface.
Mirror_Cbk.h	This header file is only included if bus mirroring is enabled (`LINIF_BUSMIRRORING == STD_ON`).
PduR_LinTp.h	This header file only included if LinTp support is enabled (`LINIF_TP_SUPPORTED == STD_ON`).
Rtm.h	This header file is only included if runtime measurement is enabled (`LINIF_RUNTIME_MEASUREMENT_SUPPORT == STD_ON`).
TP related header	TP header files are only included if LinTp support is enabled (`LINIF_TP_SUPPORTED == STD_ON`).

Table 3-4 Optional includes

3.3 Critical Sections

The MICROSAR Classic LIN Interface has code sections which are executed on interrupt level. Furthermore, the LIN interface calls LIN Driver functions on task level which do not contain any locking mechanism, so the LIN interrupts must also be disabled by the LIN Interface interrupt locking.

Additionally, 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) [9]. 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()
```

3.4 Critical Section Codes

To ensure data consistency and a correct function of the LINIF some 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_<LinIfChannel.ShortName>()` (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

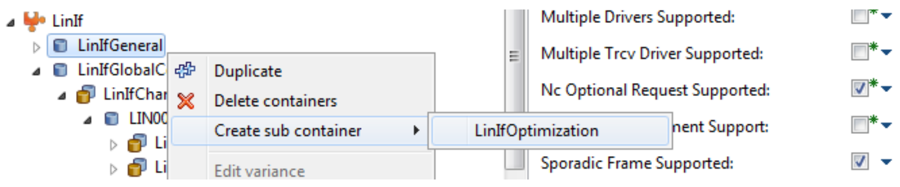
- Must lock global interrupts if `LinIf_MainFunction_<LinIfChannel.ShortName>()` can interrupt its own API functions (i.e. preemptive task system, where LINIF is running in higher prior task than LinSM and call of `LinIf_GotoSleep()` is interrupted by `LinIf_MainFunction_<LinIfChannel.ShortName>()`).

LINIF_EXCLUSIVE_AREA_2 (only used by LIN slave nodes)

- Must lock interrupts of the LIN hardware.
- Must lock global interrupts if API functions of the LINIF can interrupt the LINIF callback functions which are called by LIN driver in interrupt context.

3.5 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 3-2:



Multiple Drivers Supported: ☐
Multiple Trcv Driver Supported: ☐
Nc Optional Request Supported: ✓
nent Support: ☐
Sporadic Frame Supported: ✓

Figure 3-2 Activation of ComStackLib optimizations

The same applies for the LinTpGeneral container. For further information of the particular options, please refer to [12]. By default, all optimization is switched off.

3.6 Integration hints for special use cases

3.6.1 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 [8] for further details.

3.6.2 Main function 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_<LinIfChannel.ShortName>()` triggers the transmission of a LIN frame header at the beginning of a schedule slot, as illustrated in following figure.

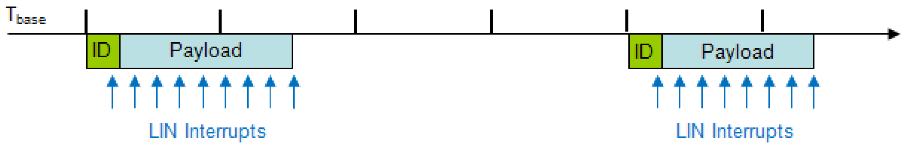


Figure 3-3 LIN frame processing in relation to main function cycles

As consequence, it turned out to be useful to execute the channel main function `LinIf_MainFunction_<LinIfChannel.ShortName>()` 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 task cycle settings containing a LinIf main function.

Reference
If a MICROSAR Classic RTE and OS are used, this setting can be comfortable configured in DaVinci Configurator, see [11] for further details.

3.6.3 Synchronization of Schedule Tables (LIN master)

Note
This chapter is only applicable to LIN master nodes.

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 all LinIf_MainFunction_<LinIfChannel.ShortName>().

It is highly recommended to enable parameter ‘LinIfSeparateMainFunctionPostProcessing’ to minimize the jitter between the main functions, see 2.5 for further details.

3.6.4 3rd party LIN driver usage

The LinIf provides dedicated support to use not only MICROSAR LIN drivers, but also AUTOSAR compatible 3rd party LIN driver. LIN driver modules implemented according to following AUTOSAR versions are supported:

- AUTOSAR 4.3.1
- AUTOSAR 4.4.0
- AUTOSAR 21-11

The compatibility to different AUTOSAR versions is configurable for each LIN channel.

Using the configuration parameter 'LinIfDrvArVersion', the used AUTOSAR version respectively MICROSAR can be selected. This is the only parameter that needs to be configured in container 'LinIfLinDriverAutosarCompatibility' by the user, the generator validates the correct setting of the concrete configuration options.

In the following, configuration options are described that have an impact on the AUTOSAR version of the underlying LIN driver:

LIN driver startup state (parameter 'LinIfLinDriverStartupState'):

Configures the actual network state of the used LIN driver after it is initialized.

- MICROSAR LIN driver: Shall be set to NORMAL.
- AUTOSAR LIN driver: Shall be set to SLEEP.
- See also chapter 2.3.

LIN driver status call after wakeup transmission (parameter 'LinIfStatusCallAfterWakeupTx'):

Configures if the LIN driver is requested for its status via `Lin_GetStatus()` after a wakeup frame transmission was requested and the maximum wakeup transmission time is elapsed. This parameter is only relevant for LIN master nodes.

- MICROSAR LIN driver: Shall be set to True.
- AUTOSAR LIN driver: Shall be set to False.

LIN driver legacy frame response type (parameter 'LinIfUseLegacyFrameResponseType'):

Configures if the definition of Lin_FrameResponseType according to AUTOSAR <= 4.3.1 is used or the definition from AUTOSAR 4.4 onwards.

- AUTOSAR 4.3.1 LIN driver: Shall be set to True.
- MICROSAR, AUTOSAR 4.4.0 and R21-11 LIN driver: Shall be set to False.

Even if the parameter LinIfUseLegacyFrameResponseType is configured per LinIf channel, it is a global configuration switch. Only one definition of Lin_FrameResponseType can be used at the same time by default, see following integration note how to support different frame response type definitions in the same configuration.

Caution
A combined usage of AUTOSAR 4.3.1 and MICROSAR LIN drivers in one configuration is not supported out-of-box due to the two incompatible definitions of Lin_FrameResponseType.

However, the use case can be supported via adaptations during integration. The recommended solution is to disable the 'LinIfUseLegacyFrameResponseType' parameter and provide following modified definition of Lin_FrameResponseType in Lin_GeneralTypes.h which covers both types:

```
/* Lin_FrameResponseType has changed between AUTOSAR 4.3.1 and AUTOSAR 4.4.0.
   Below definition contains both variants in one enumeration.
   This is explicitly allowed:
   ISO/IEC 9899:1990, 6.5.2.2
   ISO/IEC 9899:1999, 6.7.2.2 */
/*! This type is used to specify whether the frame processor is required to
    transmit the response part of the LIN frame. */
typedef enum Lin_FrameResponseTypeTag
{
  /* Lin_FrameResponseType according to AUTOSAR <= 4.3.1 */
  LIN_MASTER_RESPONSE = 0, /*<! Response is generated from this (master) node */
  LIN_SLAVE_RESPONSE  = 1, /*<! Response is generated from a remote slave node */
  LIN_SLAVE_TO_SLAVE  = 2, /*<! Response is generated from one slave to another
                                slave */

  /* Lin_FrameResponseType according to AUTOSAR >= 4.4.0 */
  LIN_FRAMERESPONSE_TX     = 0, /*<! Response is generated by this node. */
  LIN_FRAMERESPONSE_RX     = 1, /*<! Response is generated by another node. */
  LIN_FRAMERESPONSE_IGNORE = 2  /*<! Response is ignored by this node. */
} Lin_FrameResponseType;
```

LIN driver callback channel handle type (parameter 'LinIfChannelHandleTypeInCallbacks'):

Configures the channel handle type used by LIN driver in slave callback functions:

- LinIf_HeaderIndication
- LinIf_RxIndication
- LinIf_TxConfirmation
- LinIf_LinErrorIndication

This parameter is only relevant for LIN slave nodes

- MICROSAR LIN driver: Shall be set to CHANNEL_TYPE_LIN.
- AUTOSAR LIN driver: Shall be set to CHANNEL_TYPE_COMM.

4 API Description

For an interfaces overview please see Figure 1-2.

4.1 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 (only applicable to LIN master nodes)	N/A

Table 4-1 Type definitions

4.2 Services provided by LINIF

4.2.1 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 4-2 LinIf_Init

4.2.2 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 4-3 LinIf_InitMemory

4.2.3 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). > This function can be called even when the module is not yet initialized.	
Call context	
> Function can be called from task and interrupt context.	

Table 4-4 LinIf_GetVersion

4.2.4 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. > This function is currently only applicable to LIN master nodes. It has no effect for LIN frames that belongs to LIN slave channels.	
Call context	
> This function can be called from task context only.	

Table 4-5 LinIf_Transmit

4.2.5 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. The channel must reference a LIN master channel.
	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	
> This function is only available if LIN master nodes are supported. > This function is only applicable to LIN master channels. > If a non-existent table is referenced or the channel is in SLEEP, the function returns with E_NOT_OK.	

> Usage of exclusive areas.
Call context
> This function can be called from task context only.

Table 4-6 LinIf_ScheduleRequest

4.2.6 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	
> Function can be called from task and interrupt context.	

Table 4-7 LinIf_GotoSleep

4.2.7 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
> Function can be called from task and interrupt context.

Table 4-8 LinIf_Wakeup

4.2.8 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	
> This function can be called from task context only.	

Table 4-9 LinIf_SetTrcvMode

4.2.9 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	

> This function can be called from task context only.

Table 4-10 LinIf_GetTrcvMode

4.2.10 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	
> This function can be called from task context only.	

Table 4-11 LinIf_GetTrcvWakeupReason

4.2.11 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	
> This function can be called from task context only.	

Table 4-12 LinIf_SetTrcvWakeupMode

4.2.12 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	
> This function can be called from task context only.	

Table 4-13 LinIf_CancelTransmit

4.2.13 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_ReturnType`	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	
> Function can be called from task and interrupt context.	

Table 4-14 LinIf_ CheckWakeup

4.2.14 LinIf_MainFunction_<LinIfChannel.ShortName>

Prototype	
`void LinIf_MainFunction_<LinIfChannel.ShortName> (void)`	
Parameter	
-	-
Return code	
-	-
Functional Description	
This function has to be called with the configured 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. It processes timers and states of the module. For LIN master channels, it performs the schedule table handling and processes the schedule slots. For further information about the call cycle refer to chapter 2.5.	
Particularities and Limitations	
> Usage of exclusive areas for each channel specific handling.	
Call context	
> This function can be called from task context only.	

Table 4-15 LinIf_MainFunction_<LinIfChannel.ShortName>

4.2.15 LinIf_MainFunctionPostProcessing_<LinIfChannel.ShortName>

Prototype	
`void LinIf_MainFunctionPostProcessing_<LinIfChannel.ShortName> (void)`	
Parameter	
-	-
Return code	
-	-
Functional Description	
This function has to be called with the configured 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. It processes all upper layer interactions and notifications for one LIN master channel. For further information about the call cycle refer to chapter 2.5.	
Particularities and Limitations	
> No usage of exclusive areas. > This function is only available if LIN master nodes are supported. > This function is only applicable to LIN master channels. > This function is only available if separate main function post processing is enabled.	
Call context	
> This function can be called from task context only.	

Table 4-16 LinIf_MainFunctionPostProcessing_<LinIfChannel.ShortName>

4.2.16 LinIf_GetScheduleInfo

Prototype	
`void LinIf_GetScheduleInfo(NetworkHandleType Channel, LinIf_ScheduleInfoType ScheduleInfo)`	
Parameter	
`Channel`	The channel on which to request the schedule table information. The channel must reference a LIN master channel.
`ScheduleInfo`	Struct pointer to store the schedule information to.
Return code	
`Std_ReturnType`	Result of function can be: `E_OK, E_NOT_OK`
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 LIN master nodes are supported. > This function is only applicable to LIN master channels. > This function is only available if schedule info API is enabled.	
Call context	
> This function can be called from task context only.	

Table 4-17 LinIf_GetScheduleInfo

4.2.17 LinIf_EnableBusMirroring

Prototype	
`Std_ReturnType LinIf_EnableBusMirroring (NetworkHandleType Channel, boolean MirroringActive)`	
Parameter	
`Channel`	The channel on which to enable/disable mirroring.
`MirroringActive`	Requested activation state of bus mirroring.
Return code	
`Std_ReturnType`	Result of function can be: `E_OK, E_NOT_OK`
Functional Description	
This functions requests to change the mirror mode of a LIN channel to either enabled or disabled.	
Particularities and Limitations	
> Usage of exclusive areas. > This function is only available if bus mirroring is enabled.	
Call context	

> This function can be called from task context only.

Table 4-18 LinIf_EnableBusMirroring

4.2.18 LinIf_GetConfiguredNAD

Prototype	
`Std_ReturnType LinIf_GetConfiguredNAD (NetworkHandleType Channel, uint8* NAD)`	
Parameter	
`Channel`	The channel on which to get the configured NAD. The channel must reference a LIN slave channel.
`NAD`	Pointer to location to store the NAD. Parameter must not be NULL. After successful return, it contains the configured NAD value.It is only valid if E_OK is returned from the function call.
Return code	
`Std_ReturnType`	Result of function can be: `E_OK, E_NOT_OK`
Functional Description	
Returns the currently configured NAD of the slave node.	
Particularities and Limitations	
> Usage of exclusive areas. > This function is only available if LIN slave nodes are supported. > This function is only applicable to LIN slave channels.	
Call context	
> This function can be called from task context only.	

Table 4-19 LinIf_GetConfiguredNAD

4.2.19 LinIf_SetConfiguredNAD

Prototype	
`Std_ReturnType LinIf_SetConfiguredNAD (NetworkHandleType Channel, uint8 NAD)`	
Parameter	
`Channel`	The channel on which to set the configured NAD. The channel must reference a LIN slave channel.
`NAD`	NAD to set as new configured NAD for slave node.
Return code	
`Std_ReturnType`	Result of function can be: `E_OK, E_NOT_OK`
Functional Description	
Sets the currently configured NAD of the slave node to the provided NAD value.	
Particularities and Limitations	
> Usage of exclusive areas. > This function is only available if LIN slave nodes are supported.	

> This function is only applicable to LIN slave channels.
Call context
> This function can be called from task context only.

Table 4-20 LinIf_SetConfiguredNAD

4.2.20 LinIf_GetPIDTable

Prototype	
`Std_ReturnType LinIf_GetPIDTable (NetworkHandleType Channel, Lin_FramePidType *PidBuffer, uint8 *PidBufferLength)`	
Parameter	
`Channel`	The channel on which to set the configured NAD. The channel must reference a LIN slave channel.
`PidBuffer`	Pointer to buffer to which the current configured PIDs shall be copied to.
`PidBufferLength`	Length of provided buffer, pointed to by PidBuffer. After successful return it contains the number of copied PID values. If zero is provided, after successful return it contains the number of configured PID values, without updating the buffer.
Return code	
`Std_ReturnType`	Result of function can be: `E_OK, E_NOT_OK`
Functional Description	
Return the configured PIDs of all relevant frames of the slave node. The order of the returned configured PIDs is ascending, corresponding to frame index. The PIDs for the master request frame and slave response frame are not included.	
Particularities and Limitations	
> Usage of exclusive areas. > This function is only available if LIN slave nodes are supported. > This function is only applicable to LIN slave channels.	
Call context	
> This function can be called from task context only.	

Table 4-21 LinIf_GetPIDTable

4.2.21 LinIf_SetPIDTable

Prototype	
`Std_ReturnType LinIf_SetPIDTable (NetworkHandleType Channel, const Lin_FramePidType *PidBuffer, uint8 PidBufferLength)`	
Parameter	
`Channel`	The channel on which to set the configured NAD. The channel must reference a LIN slave channel.

PidBuffer	Pointer to buffer which contains the PID values to configure.
PidBufferLength	Number of PID values in the provided buffer.
Return code	
Std_ReturnType	Result of function can be: `E_OK, E_NOT_OK`
Functional Description	
Sets the configured PIDs of all relevant frames of the slave node. The configured PIDs of the slave node are updated with the PIDs of the provided buffer. The order of configured PIDs shall be ascending, corresponding to frame index. the master request frame and slave response frame are not included.	
Particularities and Limitations	
> Usage of exclusive areas. > This function is only available if LIN slave nodes are supported. > This function is only applicable to LIN slave channels.	
Call context	
> This function can be called from task context only.	

Table 4-22 LinIf_SetPIDTable

4.2.22 LinIf_SetLinProductIdentification

Prototype	
`Std_ReturnType LinIf_SetLinProductIdentification (NetworkHandleType Channel, LinIf_ProductIdentType ProductIdentKey, uint16 Value)`	
Parameter	
Channel	The channel on which to set the configured NAD. The channel must reference a LIN slave channel.
ProductIdentKey	Identifier of the property of the LIN product identification to update. Valid values are: LINIF_LINPRODIDENT_SUPPLIER_ID: Updates the Supplier ID LINIF_LINPRODIDENT_FUNCTION_ID: Updates the Function ID LINIF_LINPRODIDENT_VARIANT_ID: Updates the Variant ID
Value	New value to set as LIN product identification, selected by ProductIdentKey. Supplier ID: Valid range is [0, 0x7FFF]. Function ID: Valid range is [0, 0xFFFF]. Variant ID: Valid range is [0, 0xFF]. Only LSB of parameter Value is used.
Return code	
Std_ReturnType	Result of function can be: `E_OK, E_NOT_OK`
Functional Description	
Sets the LIN product identification values. Updates the LIN product identification, consisting of Supplier ID, Function ID and Variant ID, during runtime. Allows overwriting the static configured LIN product identification.	

Particularities and Limitations
> Usage of exclusive areas. > This function is only available if LIN slave nodes are supported and configurable product identification is enabled. > This function is only applicable to LIN slave channels.
Call context
> This function can be called from task context only.

Table 4-23 LinIf_SetLinProductIdentification

4.2.23 LinIf_SetJ2602StatusByteResetBit

Prototype	
`Std_ReturnType LinIf_SetJ2602StatusByteResetBit (NetworkHandleType Channel, boolean TriggerResponse)`	
Parameter	
`Channel`	The channel on which to set the reset state in the J2602 error field of the status byte. The channel must reference a LIN slave channel.
`TriggerResponse`	TRUE: Requests the positive response transmission for the reset request. FALSE: No response is transmitted for the reset request
Return code	
`Std_ReturnType`	Result of function can be: `E_OK`, `E_NOT_OK`
Functional Description	
Requests to set reset state bit active in the J2602 status byte. After a software reset has been performed after reception of a Targeted Reset service, the application can set the reset state bit afterwards using this function. Optionally, a positive request is marked as pending and transmitted when the next slave response header is received. This is mandatory for a previous physical Targeted Reset request while no response shall be sent for broadcast reset command.	
Particularities and Limitations	
> Usage of exclusive areas. > This function is only available if LIN slave nodes are supported and LIN protocol version SAE J2602 (2012) is configured. > This function is only applicable to LIN slave channels.	
Call context	
> This function can be called from task context only.	

Table 4-24 LinIf_SetJ2602StatusByteResetBit

4.2.24 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 4-25 LinTp_Init

4.2.25 LinTp_Transmit

Prototype	
`Std_ReturnType LinTp_Transmit(PduIdType LinTpTxSduId, const PduInfoType * LinTpTxInfoPtr)`	
Parameter	

`LinTpTxSduId`
 	
`LinTpTxInfoPtr`
	This parameter contains the unique identifier of the NSDU (TP message) to be transmitted.
	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	
> This function can be called from task context only.	

Table 4-26 LinTp_Transmit

4.2.26 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	
> The function is only available if enabled by parameter 'LinIfVersionInfoApi' (LINTP_VERSION_INFO_API == ON). > This function can be called even when the module is not yet initialized.	
Call context	
> Function can be called from task and interrupt context.	

Table 4-27 LinTp_GetVersionInfo

4.2.27 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	
> This function can be called from task context only.	

Table 4-28 LinTp_Shutdown

4.2.28 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	
> This function can be called from task context only.	

Table 4-29 LinTp_ChangeParameter

4.2.29 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	
> This function can be called from task context only.	

Table 4-30 LinTp_CancelTransmit

4.2.30 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
> This function can be called from task context only.

Table 4-31 LinTp_CancelReceive

4.3 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
Com	Com_SendSignal()

DET	Det_ReportError()
	Det_ReportRuntimeError()

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()

Component	API
	Rtm_Stop()
LinSm	LinSM_ScheduleEndNotification()
Mirror	Mirror_ReportLinFrame()

Table 4-32 Services used by the LINIF

4.4 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.

4.4.1 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	
> Usage of exclusive areas.	
Call context	
> Function can be called from task and interrupt context.	

Table 4-33 LinIf_WakeupConfirmation

4.4.2 LinIf_HeaderIndication

Prototype	
`void LinIf_HeaderIndication (NetworkHandleType Channel, Lin_PduType *PduPtr)`	
Parameter	
`Channel`	CHANNEL_TYPE_LIN : LIN driver channel handle. The channel must reference a LIN slave channel. CHANNEL_TYPE_COMM: ComM channel handle. The channel must reference a LIN slave channel.
`PduPtr`	Pointer to PDU providing the received PID and pointer to the SDU data buffer. Upon return, the remaining members are set as out parameters.

Return code	
–	-
Functional Description	
The LIN driver calls this function to report a received LIN header and to provide the received PID.	
Particularities and Limitations	
> Usage of exclusive areas. > This function is only available if LIN slave nodes are supported. > This function is only applicable to LIN slave channels. > The channel handle type must be configured according to 3.6.4.	
Call context	
> Function can be called from task and interrupt context.	

Table 4-34 LinIf_HeaderIndication

4.4.3 LinIf_RxIndication

Prototype	
`void LinIf_RxIndication (NetworkHandleType Channel, uint8 *SduPtr)`	
Parameter	
`Channel`	CHANNEL_TYPE_LIN : LIN driver channel handle. The channel must reference a LIN slave channel. CHANNEL_TYPE_COMM: ComM channel handle. The channel must reference a LIN slave channel.
`SduPtr`	Pointer to SDU providing the received data bytes.
Return code	
–	-
Functional Description	
The LIN driver calls this function to report a successfully received response and to provide the reception data.	
Particularities and Limitations	
> Usage of exclusive areas. > This function is only available if LIN slave nodes are supported. > This function is only applicable to LIN slave channels. > The channel handle type must be configured according to 3.6.4.	
Call context	
> Function can be called from task and interrupt context.	

Table 4-35 LinIf_RxIndication

4.4.4 LinIf_TxConfirmation

Prototype
`void LinIf_TxConfirmation (NetworkHandleType Channel)`

Parameter	
`Channel`	CHANNEL_TYPE_LIN : LIN driver channel handle. The channel must reference a LIN slave channel. CHANNEL_TYPE_COMM: ComM channel handle. The channel must reference a LIN slave channel.
Return code	
–	-
Functional Description	
The LIN driver calls this function to report a successfully transmitted response.	
Particularities and Limitations	
> Usage of exclusive areas. > This function is only available if LIN slave nodes are supported. > This function is only applicable to LIN slave channels. > The channel handle type must be configured according to 3.6.4.	
Call context	
> Function can be called from task and interrupt context.	

Table 4-36 LinIf_TxConfirmation

4.4.5 LinIf_LinErrorIndication

Prototype	
`void LinIf_LinErrorIndication (NetworkHandleType Channel, Lin_SlaveErrorType ErrorStatus)`	
Parameter	
`Channel`	CHANNEL_TYPE_LIN : LIN driver channel handle. The channel must reference a LIN slave channel. CHANNEL_TYPE_COMM: ComM channel handle. The channel must reference a LIN slave channel.
`ErrorStatus`	Type of detected error.
Return code	
–	-
Functional Description	
The LIN driver calls this function to report report a detected error event during header or response processing.	
Particularities and Limitations	
> Usage of exclusive areas. > This function is only available if LIN slave nodes are supported. > This function is only applicable to LIN slave channels. > The channel handle type must be configured according to 3.6.4.	
Call context	
> Function can be called from task and interrupt context.	

Table 4-37 LinIf_LinErrorIndication

4.5 Configurable Interfaces

4.5.1 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.

4.5.1.1 <User>_ScheduleRequestConfirmation

Prototype	
`void <User>_ScheduleRequestConfirmation (NetworkHandleType channel, LinIf_SchhandleType schedule)`	
Parameter	
`channel`	The channel on which a schedule table change was performed.
`schedule`	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 4-38 <User>_ScheduleRequestConfirmation

4.5.1.2 <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 4-39 <User>_GotoSleepConfirmation

4.5.1.3 <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 4-40 <User>_WakeupConfirmation

4.5.1.4 <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 4-41 <User>_TriggerTransmit

4.5.1.5 <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 4-42 <User>_TxConfirmation

4.5.1.6 <User>_RxIndication

Prototype	
`void <User>_RxIndication (PduIdType RxPduId, PduInfoType* PduInfoPtr)`	
Parameter	
`TxPduId`	ID of the SDU that has been received.
`PduInfoPtr`	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 4-43 <User>_RxIndication

4.5.2 Callout Functions

At its configurable interfaces the LINIF defines callout functions. The declarations of the callout functions are provided by the BSW module, i.e. the LINIF. It is the integrator's task to provide the corresponding function definitions. The definitions of the callouts can be adjusted to the system's needs. The LINIF callout function declarations are described in the following tables:

4.5.2.1 <ResponseErrorSignalChangedCallout>

Prototype
`void <ResponseErrorSignalChangedCallout> (NetworkHandleType Channel, boolean RespErrSigValue)`

Parameter	
`Channel`	The channel on which the response error signal value changed.
`RespErrSigValue`	Current value of the response error signal: TRUE: Response error signal is set. FALSE: Response error signal is cleared.
Return code	
–	-
Functional Description	
This callout function is called each time the value of the response error signal has changed.	
Particularities and Limitations	
> This function is only available if LIN slave nodes are supported and a response error signal changed callout function is configured.	
Call context	
> Called on interrupt level inside exclusive areas.	

Table 4-44 <ResponseErrorSignalChangedCallout>

4.5.2.2 <SaveConfigurationCallout>

Prototype	
`boolean <SaveConfigurationCallout> (NetworkHandleType Channel)`	
Parameter	
`Channel`	The channel on which the SaveConfiguration request has been received.
Return code	
`boolean`	TRUE: A positive response shall be transmitted. FALSE: No response shall be transmitted.
Functional Description	
This callout function is called when a SaveConfiguration request addressing this node has been received.	
Particularities and Limitations	
> This function is only available if LIN slave nodes are supported and a SaveConfiguration callout function is configured.	
Call context	
> Called on interrupt level inside exclusive areas.	

Table 4-45 <SaveConfigurationCallout>

4.6 Callouts

The LINIF defines callout functions with static interface definitions. The declarations of the callout functions are provided by the BSW module, i.e. the LINIF. It is the integrator's task to provide the corresponding function definitions. The LINIF callout function declarations are described in the following tables:

4.6.1 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`	
> This function is only applicable to LIN master channels.	
Call context	
> Called on task level inside exclusive areas.	

Table 4-46 Appl_LinIfGetLinStatus

4.6.2 Appl_LinIfJ2602TargetedReset

Prototype	
`LinIf_J2602TargetedResetReturnType Appl_LinIfJ2602TargetedReset (NetworkHandleType Channel, uint8 Nad)`	
Parameter	
`Channel`	The channel on which a frame was processed.
`Nad`	NAD which is received in request.
Return code	
`LinIf_J2602TargetedReset_PosResp`	A positive response requested. In case of a broadcast request, the response is suppressed.
`LinIf_J2602TargetedReset_NegResp`	A negative response requested. In case of a broadcast request, the response is suppressed.
`LinIf_J2602TargetedReset_NoResp`	No response requested. It is assumed that the response is triggered by application later. Note that according to SAE J2602 a response shall be sent for a physical request in any case.

Functional Description
This callback function is called after reception of a J2602 Targeted Reset request addressing this slave node. The application decides with the return value the type of the response: either a positive response if the request is accepted, a negative response if the request is declined or no response at all. The latter case is intended to use if a reset is performed in software and the response is sent after the reset (either a positive or negative response is mandatory). The NAD of the request is provided to distinguish physical and broadcast requests. In case of a broadcast reset request, no response is sent at all, independent of the provided return value. If a positive response is requested, then the reset bit in the J2602 status byte is set.
Particularities and Limitations
> This function is only available if LIN slave nodes are supported and LIN protocol version SAE J2602 (2012) is configured.
Call context
> Called on interrupt level inside exclusive areas.

Table 4-47 Appl_LinIfJ2602TargetedReset

4.6.3 Appl_LinIfJ2602StatusByteProcessFrame

Prototype	
`boolean Appl_LinIfJ2602StatusByteProcessFrame (NetworkHandleType Channel, uint8 Pid)`	
Parameter	
`Channel`	The channel on which a frame was transmitted.
`Pid`	PID of transmitted frame.
Return code	
`TRUE`	The frame contains a J2602 status byte and the status error field shall be updated by the LinIf accordingly. This is the default case.
`FALSE`	The frame does not contain a J2602 status byte. The handling of the status error field shall be skipped by LinIf for this frame.
Functional Description	
Requests the application if the current frame contains a J2602 status byte. In general the LinIf assumes, according to the SAE J2602 specification, that each unconditional Tx frame includes a J2602 status error field. So after successful transmission of an unconditional frame, a possible pending error status is cleared and the J2602 status error field is updated accordingly. However, in special configurations, some unconditional Tx frames might not contain a J2602 status error field. This callback function can be used to inform if a transmitted unconditional frame contains a J2602 status error field or does not contain it. In latter case, the update of the status error field is skipped. The integrator must ensure that the configuration of the J2602 status byte references in the LinIf coincides with the implementation of this callback.	

Particularities and Limitations
> This function is only available if LIN slave nodes are supported and LIN protocol version SAE J2602 (2012) is configured. > This callout function has to be enabled by specifying a user config file with following content: `#define LINIF_SLAVE_J2602_STATUSBYTE_SKIP_FRAMES    STD_ON`
Call context
> Called on interrupt level inside exclusive areas.

Table 4-48 Appl_ LinIfJ2602StatusByteProcessFrame

4.6.4 Appl_LinIfReadByIdSerialNumber

Prototype	
`LinIf_ReadByIdSerialNumberReturnType Appl_LinIfReadByIdSerialNumber (NetworkHandleType Channel, uint8 Nad, uint8* ResponseData)`	
Parameter	
`Channel`	The channel on which the request was received.
`NAD`	Received NAD in request.
`ResponseData`	Pointer to an uint8 buffer with four elements. If the return value requests a positive response, this buffer must be filled by application with the 4 byte serial number to be used in the positive response. The LSB of the serial number shall be written to ResponseData[0] and the MSB to ResponseData[3]. If the return value requests a negative response, this buffer is not used and needs not to be written. Valid write access index range: [0, 3].
Return code	
`LinIf_ReadByIdSerialNumber_PosResp`	A positive response with the provided response data is requested.
`LinIf_ReadByIdSerialNumber_NegResp`	A negative response is requested.
Functional Description	
Informs the application about the reception of a ReadByIdentifier request with identifier 1. This callback function is called after reception of a ReadByIdentifier request with identifier 1 (serial number) addressing this slave node. The application decides with the return value the type of the response: Either a positive response is transmitted if the request is accepted or a negative response is transmitted if the request is declined. In case of a positive response, the application must provide the response data using the pointer ResponseData, see description of parameter `ResponseData` for usage details and example implementation below.	
Particularities and Limitations	
> This function is only available if LIN slave nodes are supported and LIN protocol version 2.0 or greater is configured and if configuration parameter 'ReadBy Identifier Serial Number Supported' is enabled.	
Call context	
> Called on interrupt level inside exclusive areas.	

Table 4-49 Appl_ LinIfReadByIdSerialNumber

Example

Following example shows an example implementation which depending on some condition either transmits a positive response with serial number 0x11223344 or a negative response:

```
FUNC(LinIf_ReadByIdSerialNumberReturnType, LINIF_CODE)
     Appl_LinIfReadByIdSerialNumber
(
  VAR(NetworkHandleType, AUTOMATIC)          Channel,
  VAR(uint8, AUTOMATIC)                      Nad,
  P2VAR(uint8, AUTOMATIC, LINIF_APPL_VAR)     ResponseData
)
{
  if ( <your_condition> )
  {
    /* send positive response and serial number 0x44332211 */
    ResponseData[0] = 0x11u;
    ResponseData[1] = 0x22u;
    ResponseData[2] = 0x33u;
    ResponseData[3] = 0x44u;
    return LinIf_ReadByIdSerialNumber_PosResp;
  }
  else
  {
    /* send negative response */
    return LinIf_ReadByIdSerialNumber_NegResp;
  }
}
```

4.6.5 Appl_LinIfReadByIdUserDefined

Prototype	
`LinIf_ReadByIdUserDefinedReturnType Appl_LinIfReadByIdUserDefined (NetworkHandleType Channel, uint8 Nad, uint8 Id, uint8* ResponseData, uint8* ResponseLength)`	
Parameter	
`Channel`	The channel on which the request was received.
`NAD`	Received NAD in request.
`Id`	Received identifier, possible range: [32, 63].
`ResponseData`	Pointer to an uint8 buffer with five elements. If the return value requests a positive response, this buffer must be filled by the application with the user defined data bytes. The actual response length (PCI) must be written to parameter `ResponseLength`. Even if the response length is less than 5, it is recommended to fill all 5 buffer elements to have defined response data on the bus using fill byte 0xFF. Valid write access index range: [0, 4].

`ResponseLength`	Pointer to an uint8 variable to specify the response length. If the return value requests a positive response, this value must be set to the actual number of data bytes written into the buffer provided via pointer ResponseData. Valid range of response length: [1, 5].
Return code	
`LinIf_ReadByIdUserDefined_PosResp`	A positive response with the provided response data and length is requested.
`LinIf_ReadByIdUserDefined_NegResp`	A negative response is requested.
`LinIf_ReadByIdUserDefined_NoResp`	No response is requested. Note that according to the LIN protocol, there should always be a response to this request (positive or negative), but to handle all possible use cases for user defined requests, also no response is supported.
Functional Description	
Informs the application about the reception of a ReadByIdentifier request with identifier [32, 63]. This callback function is called after reception of a ReadByIdentifier request with identifier in range [32, 63] (user defined) addressing this slave node. The application decides with the return value the type of the response: either a positive response if the request is accepted , a negative response if the request is declined or no response at all. In case of a positive response, the application must also provide the response data and length, see description of parameters `ResponseData` and `ResponseLength` for usage details and example implementation below.	
Particularities and Limitations	
> This function is only available if LIN slave nodes are supported and LIN protocol version 2.0 or greater is configured and if configuration parameter ‘ReadBy Identifier User Defined Supported’ is enabled.	
Call context	
> Called on interrupt level inside exclusive areas.	

Table 4-50 Appl_ LinIfReadByIdUserDefined

Example
Following example shows an example implementation which transmits for identifier 42 a positive response with data 0xA1, 0xA2, 0xA3, 0xA4 and length 4, else a negative response.

```
FUNC(LinIf_ReadByIdUserDefinedReturnType, LINIF_CODE)
     Appl_LinIfReadByIdUserDefined
(
  VAR(NetworkHandleType, AUTOMATIC)           Channel,
  VAR(uint8, AUTOMATIC)                       Nad,
  VAR(uint8, AUTOMATIC)                       Id,
  P2VAR(uint8, AUTOMATIC, LINIF_APPL_VAR)     ResponseData,
  P2VAR(uint8, AUTOMATIC, LINIF_APPL_VAR)     ResponseLength
)
{
  if ( Id == 42 )
  {
    ResponseData[0] = 0xA1u;
    ResponseData[1] = 0xA2u;
    ResponseData[2] = 0xA3u;
    ResponseData[3] = 0xA4u;
    ResponseData[4] = 0xFFu; /* stuffing byte */
    *ResponseLength = 0x04u;
    return LinIf_ReadByIdUserDefined_PosResp;
  }
  else
  {
    return LinIf_ReadByIdUserDefined_NegResp;
  }
}
```

4.6.6 Appl_LinIfDataDump

Prototype	
`LinIf_DataDumpReturnType Appl_LinIfDataDump (NetworkHandleType Channel, uint8 Nad, const uint8* RequestData, uint8* ResponseData)`	
Parameter	
`Channel`	The channel on which the request was received.
`NAD`	Received NAD in request.
`RequestData`	Pointer to an uint8 buffer with five elements which contains the received five user defined data bytes of the request. Valid read access index range: [0, 4]
`ResponseData`	Pointer to an uint8 buffer with five elements. If the return value requests a positive response, this buffer must be filled by the application with the user defined data bytes to be used in the positive response. If the return value requests no response, this buffer is not further used and needs not to be written. Valid write access index range: [0, 4]
Return code	
`LinIf_DataDump_PosResp`	A positive response with the provided response data is requested.

`LinIf_DataDump_NoResp`	No response is requested.
Functional Description	
Informs the application about the reception of a DataDump request. This callback function is called after reception of a DataDump request addressing this slave node. The application decides with the return value the type of the response: either a positive response if the request is accepted or no response is transmitted if the request is declined. In case of a positive response, the application must provide the response data, see the pointer parameter `ResponseData` for usage details and example implementation below.	
Particularities and Limitations	
> This function is only available if LIN slave nodes are supported and LIN protocol version 2.0 or greater is configured and if configuration parameter 'Data Dump Supported' is enabled.	
Call context	
> Called on interrupt level inside exclusive areas.	

Table 4-51 Appl_ LinIfDataDump

Example

Following example shows an example implementation which transmits a positive response with data 0xA1, 0xA2, 0xA3, 0xA4 and 0xA5 if the request data contains 0x01 in the first byte, else no response.

```
FUNC(LinIf_DataDumpReturnType, LINIF_CODE) Appl_LinIfDataDump
(
  VAR(NetworkHandleType, AUTOMATIC)          Channel,
  VAR(uint8, AUTOMATIC)                      Nad,
  P2CONST(uint8, AUTOMATIC, LINIF_APPL_VAR) RequestData,
  P2VAR(uint8, AUTOMATIC, LINIF_APPL_VAR)   ResponseData
)
{
  if (RequestData[0] == 0x01u)
  {
    ResponseData[0] = 0xA1u;
    ResponseData[1] = 0xA2u;
    ResponseData[2] = 0xA3u;
    ResponseData[3] = 0xA4u;
    ResponseData[4] = 0xA5u;
    return LinIf_DataDump_PosResp;
  }
  else
  {
    return LinIf_DataDump_NoResp;
  }
}
```

4.6.7 Appl_LinIfAutoAddressing

Prototype
`LinIf_AutoAddressingReturnType Appl_LinIfAutoAddressing (NetworkHandleType Channel, uint8 Nad, const uint8* RequestData)`

Parameter	
`Channel`	The channel on which the request was received.
`NAD`	Received NAD in request.
`RequestData`	Pointer to an uint8 buffer with five elements which contains the received five user defined data bytes of the request. Valid read access index range: [0, 4]
Return code	
`LinIf_ AutoAddressing_PosResp`	A positive response is requested.
`LinIf_ AutoAddressing_NoResp`	No response is requested.
Functional Description	
Informs the application about the reception of an AutoAddressing request. This callback function is called after reception of an AutoAddressing request. The application decides with the return value the type of the response: either no response is transmitted (this is the default case) or an optional positive response is transmitted.	
Particularities and Limitations	
> This function is only available if LIN slave nodes are supported and LIN protocol version 2.0 or greater is configured and if configuration parameter 'Auto Addressing Supported' is enabled.	
Call context	
> Called on interrupt level inside exclusive areas.	

Table 4-52 Appl_ LinIfAutoAddressing

5 Configuration

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

- Configuration in DaVinci Configurator 5

5.1 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.

6 Standard Compliance

6.1 AUTOSAR Standard Compliance

6.1.1 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 [5] the default delay is 100ms, but can be less. This value can be configured. (LIN master)
Wakeup Delay External	Added	No wakeup delay had been specified after an external wakeup was detected. According to [5] the default delay is 100ms, but can be less. This value can be configured. (LIN master)
Support Sporadic Frame Handling	Added	Code and runtime optimization in case sporadic frames are not used. (LIN master)
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	Parameter not used for LIN master nodes: Optional NAD services are always supported by Vector LINIF if configured as schedule table commands.
LinIfTrcvWakeupNotificat ion	Removed	Parameter is unused and removed according to AR Bugzilla #52901 (AR4-569)
Schedule End Notification	Added	Support for notification callback to LinSM if current schedule table is run through. (LIN master)
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 (LIN master)
Safe Bsw Checks	Added	Support for component-specific SafeBSW configuration.
Schedule Info Api	Added	Activation of optional Schedule Information API. (LIN master)
Schedule Change Next TimeBase	Added	According to AUTOSAR 4.3.0. Configuration of point in time of actual schedule table switch. (LIN master)

Bus mirroring	Added	Support for bus mirroring, according to AUTOSAR 4.4
LIN product identification configurable	Added	Optional support for LinIf_SetLinProductIdentification API. (LIN slave)
Message ID	Added	Frame message ID added to support LIN 2.0 slave nodes. (LIN slave)
Database information support	Added	Added parameter and container to read database source and version (ReadByIdentifier service with identifier 3) (LIN slave)
Wakeup confirmation timeout period	Added	Support for dedicated timeout duration when waiting for wakeup confirmation before indicating sleep mode (LIN slave)
LIN protocol version	Range enhanced	Added enumeration value SAE J2602 2012
Response error signal	Multiplicity	Multiplicity of response error signal reference changed from 0:1 to 0:n.
ReadByIdentifier Serial Number Supported	Added	Support for ReadByIdentifier node identification command with identifier 1 (serial number) (LIN slave)
ReadByIdentifier User Defined Supported	Added	Support for ReadByIdentifier node identification command with identifier in range [32, 63] (user defined) (LIN slave)
Data Dump Supported	Added	Support for DataDump node identification command (LIN slave)
Auto Addressing Supported	Added	Support for AutoAddressing node identification command (LIN slave)
Separate MainFunction PostProcessing	Added	Support for splitted main functions for each LinIf channel (LIN master)
LIN Driver Startup State	Added	Support to handle different network states of the LIN driver after initialization. Relevant to support LIN drivers of different AUTOSAR versions.
GetStatus call after wakeup transmission	Added	Configures if the LIN driver is requested for its status after wakeup frame transmission. Relevant to support LIN drivers of different AUTOSAR versions.
Use Legacy Frame Response Type	Added	Configures if the legacy definition of Lin_FrameResponseType is used
LIN Driver AR Version	Added	Configures the AUTOSAR version of the used LIN driver.
Channel Handle Type in Callbacks	Added	Configures the type of channel handle used in slave callback functions from LIN driver.

6.1.2 Deviations within Configuration Parameters for Lin Transport Protocol

Configuration Parameter	Deviation	Reason
Forward Response Pending Frames	Added	The feature of not forwarding response pending frames to PduR is a requested customer option. (LIN master)
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. (LIN master)
Broadcast Request Handling	Added	Support of extended handling of diagnostic broadcast requests. (LIN master)
Configurable NAD	Added	Feature to mark TxNSdu and RxNSdu to use configurable NAD. (LIN slave)
Meta Data Support	Added	General switch to enable meta data handling

6.1.3 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.
LinIf_EnableBusMirroring()	Additional API	Support for bus mirroring, according to AUTOSAR 4.4
LinIf_SetLinProductIdentification()	Additional API	Support for configurable LIN product identification (LIN slave)
Appl_LinIfJ2602TargetedReset()	Additional API	Optional API to inform the upper layer about the reception of a Targeted Reset service request and to configure its response.
LinIf_SetJ2602StatusByteResetBit()	Additional API	Sets the reset error state in the J2602 status byte.

Appl_LinIfJ2602StatusByteProcessFrame()	Additional API	Optional API to request the upper layer if the current frame contains the J2602 status byte.
Appl_LinIfReadByIdSerialNumber()	Additional API	Optional API to inform the upper layer about the reception of a ReadByIdentifier request with Id = 1 and to configure its response.
Appl_LinIfReadByIdUserDefined()	Additional API	Optional API to inform the upper layer about the reception of a ReadByIdentifier request with Id = [32, 63] and to configure its response.
Appl_LinIfDataDump()	Additional API	Optional API to inform the upper layer about the reception of a DataDump request and to configure its response.
Appl_LinIfAutoAddressing()	Additional API	Optional API to inform the upper layer about the reception of a AutoAddressing request and to configure its response.

6.1.4 Deviations within features

Feature	Deviation	Reason
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 [5].
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 [10].
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. 3.2). 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.
Bus mirroring	Added	Support of mirroring unconditional frames (Rx and Tx), event-triggered frames and sporadic frames
Configurable LIN product identification	Added	Support to change the LIN product identification (Supplier ID, Function ID, Variant ID) during runtime
ReadbyIdentifier with identifier 3	Added	Support for ReadByIdentifier service with identifer 3 to retrieve database source and version (LIN slave)
AssignFrameId 2.0	Added	Support of AssignFrameId LIN 2.0 node configuration service and ReadbyIdentifier service with identifier [16, 31] (LIN slave)
Configurable NAD in LINTP	Added	Support to handle Rx / Tx connections with current configured NAD (LIN slave)
Event-triggered frames	Limited	Event-triggered frames are only support for LIN master nodes. The support for event-triggered frames on LIN slave channels is not supported.
Wakeup confirmation timeout	Added	Instead of the bus idle timeout duration, a dedicated timeout value is used to supervise the duration between internal wakeup request until wakeup confirmation in slave nodes.
SAE 2602 Targeted Reset	Added	Support of SAE J2602 Targeted Reset node configuration service. (LIN slave)
SAE J2602 Status Byte	Added	Support of SAE J2602 Status Byte handling (error field). Please note the limitation that deviation of different header errors is not supported, see 2.17 for further details. (LIN slave)
ReadbyIdentifier with identifier 1	Added	Support for ReadByIdentifier service with identifer 1 (LIN slave)
ReadbyIdentifier with identifier [32, 63]	Added	Support for ReadByIdentifier service with identifer [32, 63] (LIN slave)
DataDump	Added	Support for DataDump service (LIN slave)
AutoAddressing	Added	Support for AutoAddressing service (LIN slave)
MetaData	Added	Support for Meta data in LinTp
Separate main functions	Added	Support for two main functions per LinIf channel
Support for LIN drivers of different AUTOSAR versions	Added	Support for usage of LIN drivers according to AUTOSAR versions 4.3.1, 4.4.0 and 21-11.

6.2 ISO17987 / LIN 2.x Standard Compliance

Feature	Deviation	Reason
No response error (= completely missing response part of a frame) and response error signal for LIN 2.0	Limited	In case of a no response error, the response error signal is not set. This behavior is according to LIN 2.1 or greater. In LIN 2.0, it is expected that a completely missing response sets the response error signal. This behavior is not supported, the LinIf behaves the same way as for later LIN protocol versions. The reason is that in general the setting of the response error signal is undesirably for a missing response also in LIN 2.0 networks. A common root cause for such an error is an not assembled slave node in combination with slave-to-slave communication. Note that this specific case is not tested in the LIN 2.0 slave conformance test and therefore is still passed.
NAD in range [0x80, 0xFF]	Limited	Diagnostic frames with NAD in range [0x80, 0xFF] are allocated for free usage according to [5]. AUTOSAR does not support a direct interface to diagnostic frames from LinIf. Diagnostic frame with NAD >= 0x80 are processed by the LinTp if corresponding NSdus are configured, but the layout of these frames must comply with the transport protocol data unit format, i.e must have a valid PCI field.

7 Glossary and Abbreviations

7.1 Glossary

Term	Description
MICROSAR Classic	MICROSAR solution for AUTOSAR Classic.

Table 7-1 Glossary

7.2 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 7-2 Abbreviations

8 Contact

Visit our website for more information on

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

www.vector.com