PD ISO/TR 17987-5:2016

BSI Standards Publication

Road vehicles — Local Interconnect Network (LIN)

Part 5: Application programmers interface (API)

bsi.

National foreword

This Published Document is the UK implementation of ISO/TR 17987-5:2016.

The UK participation in its preparation was entrusted to Technical Committee AUE/16, Data Communication (Road Vehicles).

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

Published by BSI Standards Limited 2016

ISBN 978 0 580 90705 0
ICS 43.040.15

Compliance with a British Standard cannot confer immunity from legal obligations.

This Published Document was published under the authority of the Standards Policy and Strategy Committee on 30 November 2016.

Amendments/corrigenda issued since publication

Date	Text affected

TECHNICAL REPORT

ISO/TR 17987-5

First edition
2016-11-15

Road vehicles — Local Interconnect Network (LIN) —

Part 5: Application programmers interface (API)

Véhicules routiers — Réseau Internet local (LIN) —

Partie 5: Interface du programmeur d'application (API)

Reference number
ISO/TR 17987-5:2016(E)

Contents

Page

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: www.iso.org/iso/foreword.html.

The committee responsible for this document is ISO/TC 22, *Road vehicles*, Subcommittee SC 31, *Data communication*.

A list of all parts in the ISO 17987 series can be found on the ISO website.

Introduction

ISO 17987 (all parts) specifies the use cases, the communication protocol and physical layer requirements of an in-vehicle communication network called Local Interconnect Network (LIN).

The LIN protocol as proposed is an automotive focused low speed Universal Asynchronous Receiver Transmitter (UART) based network. Some of the key characteristics of the LIN protocol are signal-based communication, schedule table-based frame transfer, master/slave communication with error detection, node configuration and diagnostic service communication.

The LIN protocol is for low cost automotive control applications, for example, door module and air condition systems. It serves as a communication infrastructure for low-speed control applications in vehicles by providing:

- signal-based communication to exchange information between applications in different nodes;
- bit rate support from 1 kbit/s to 20 kbit/s;
- deterministic schedule table-based frame communication;
- network management that wakes up and puts the LIN cluster into sleep mode in a controlled manner;
- status management that provides error handling and error signalling;
- transport layer that allows large amount of data to be transmitted (such as diagnostic services);
- specification of how to handle diagnostic services;
- electrical physical layer specifications;
- node description language describing properties of slave nodes;
- network description file describing behaviour of communication;
- application programmer's interface;

ISO 17987 (all parts) is based on the open systems interconnection (OSI) Basic Reference Model as specified in ISO/IEC 7498-1 which structures communication systems into seven layers.

The OSI model structures data communication into seven layers called (top down) *application layer* (layer 7), *presentation layer*, *session layer*, *transport layer*, *network layer*, *data link layer* and *physical layer* (layer 1). A subset of these layers is used in ISO 17987 (all parts).

ISO 17987 (all parts) distinguishes between the services provided by a layer to the layer above it and the protocol used by the layer to send a message between the peer entities of that layer. The reason for this distinction is to make the services, especially the application layer services and the transport layer services, reusable also for other types of networks than LIN. In this way, the protocol is hidden from the service user and it is possible to change the protocol if special system requirements demand it.

ISO 17987 (all parts) provides all documents and references required to support the implementation of the requirements related to.

- ISO 17987-1: This part provides an overview of the ISO 17987 (all parts) and structure along with the use case definitions and a common set of resources (definitions, references) for use by all subsequent parts.
- ISO 17987-2: This part specifies the requirements related to the transport protocol and the network layer requirements to transport the PDU of a message between LIN nodes.
- ISO 17987-3: This part specifies the requirements for implementations of the LIN protocol on the logical level of abstraction. Hardware-related properties are hidden in the defined constraints.

— ISO 17987-4: This part specifies the requirements for implementations of active hardware components which are necessary to interconnect the protocol implementation.

— ISO/TR 17987-5: This part specifies the LIN application programmers interface (API) and the node configuration and identification services. The node configuration and identification services are specified in the API and define how a slave node is configured and how a slave node uses the identification service.

— ISO 17987-6: This part specifies tests to check the conformance of the LIN protocol implementation according to ISO 17987-2 and ISO 17987-3. This comprises tests for the data link layer, the network layer and the transport layer.

— ISO 17987-7: This part specifies tests to check the conformance of the LIN electrical physical layer implementation (logical level of abstraction) according to ISO 17987-4.

The LIN API is a network software layer that hides the details of a LIN network configuration (e.g. how signals are mapped into certain frames) for a user making an application program for an arbitrary ECU. The user is provided an API, which is focused on the signals transported on the LIN network. A tool takes care of the step from network configuration to program code. This provides the user with configuration flexibility. The LIN API is only one possible API existing today beside others like defined for LIN master nodes in the AUTOSAR standard. Therefore, the LIN API is published as a Technical Report and all definitions given here are informative only.

Road vehicles — Local Interconnect Network (LIN) —

Part 5:
Application programmers interface (API)

1 Scope

This document has been established in order to define the LIN application programmers interface (API).

2 Normative references

There are no normative references in this document.

3 Terms, definitions and abbreviated terms

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO 17987-2 and ISO 17987-3 apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— IEC Electropedia: available at http://www.electropedia.org/

— ISO Online browsing platform: available at http://www.iso.org/obp

3.2 Symbols

|| logical OR binary operation

3.3 Abbreviated terms

API application programmers interface

ms millisecond

OSI open systems interconnection

PDU protocol data unit

RX Rx pin of the transceiver

UART universal asynchronous receiver transmitter

4 API definitions

4.1 LIN cluster generation

The LIN Description file (LDF; see ISO 17987-2) is parsed by a tool and generates a configuration for the LIN device driver. The node capability language specification (NCF) is normally not used in this process since its intention is to describe a hardware slave node, and therefore, does not need the API. See ISO 17987-2 for a description of the workflow and the roles of the LDF and NCF.

4.2 Concept of operations

4.2.1 General

The API is split in three areas

— LIN core API,

— LIN node configuration and identification API, and

— LIN transport layer API (optional).

4.2.2 LIN core API

The LIN core API handles initialization, processing and a signal based interaction between the application and the LIN core. This implies that the application does not have to bother with frames and transmission of frames. Notification exists to detect transfer of a specific frame if this is necessary, see 4.3.5. API calls to control the LIN core also exist.

Two versions exist of most of the API calls

— static calls embed the name of the signal or interface in the name of the call, and

— dynamic calls provide the signal or interface as a parameter.

NOTE The named objects (signals, schedules) defined in the LDF extends their names with the channel postfix name (see channel postfix name definition in ISO 17987-2).

4.2.3 LIN node configuration and identification API

The LIN node configuration and identification API is service-based (request/response), i.e. the application in the master node calls an API routine that transmits a request to the specified slave node and awaits a response. The slave node device driver automatically handles the service.

The behaviour of the LIN node configuration and identification API is covered in the node configuration and identification (see ISO 17987-3).

4.2.4 LIN transport layer API

The LIN transport layer is message based. Its intended use is to work as a transport layer for messages to a diagnostic message parser outside of the LIN device driver. Two exclusively alternative APIs exist, one raw that allows the application to control the contents of every frame sent and one messaged-based that performs the full transport layer function.

The behaviour of the LIN transport layer API is defined in ISO 17987-2.

4.3 API conventions

4.3.1 General

The LIN core API has a set of functions all based on the idea to give the API a separate name space, in order to minimize the risk of conflicts with existing software. All functions and types have the prefix "l_" (lowercase "L" followed by an "underscore").

Table 1 — API functions overview

Function	Description
DRIVER AND CLUSTER MANAGEMENT	
l_sys_init	Performs the initialization of the LIN core.
SIGNAL INTERACTION	
scalar signal read	Reads and returns the current value of the signal.
scalar signal write	Reads and returns the current value of the signal.
byte array read	Reads and returns the current values of the selected bytes in the signal.
byte array write	Sets the current value of the selected bytes in the signal specified by the name sss to the value specified.
NOTIFICATION	
l_flg_tst	Returns a C boolean indicating the current state of the flag specified by the name of the static API call, i.e. returns zero if the flag is cleared, non-zero otherwise.
l_flg_clr	Sets the current value of the flag specified by the name of the static API call to zero.
SCHEDULE MANAGEMENT	
l_sch_tick	Function provides a time base for scheduling.
l_sch_set	Sets up the next schedule.
INTERFACE MANAGEMENT	
l_ifc_init	Initializes the controller specified by the name, i.e. sets up internal functions such as the baud rate.
l_ifc_goto_sleep	This call requests slave nodes on the cluster connected to the interface to enter bus sleep mode by issuing one go to sleep command.
l_ifc_wake_up	The function transmits one wake up signal.
l_ifc_ioctl	This function controls functionality that is not covered by the other API calls.
l_ifc_rx	The application program is responsible for binding the interrupt and for setting the correct interface handle (if interrupt is used).
l_ifc_tx	The application program is responsible for binding the interrupt and for setting the correct interface handle (if interrupt is used).
l_ifc_aux	This function is used in a slave nodes to synchronize to the break field/sync byte field sequence transmitted by the master node.
l_ifc_read_status	This function returns the status of the previous communication.

Table 1 *(continued)*

Function	Description
USER PROVIDED CALL-OUTS	
l_sys_irq_disable	The user implementation of this function achieves a state in which no interrupts from the LIN communication occurs.
l_sys_irq_restore	The user implementation of this function recovers the previous configured interrupt level.
NODE CONFIGURATION	
ld_is_ready	This call returns the status of the last requested configuration service.
ld_check_response	This call returns the result of the last node configuration service.
ld_assign_frame_id_range	This call assigns the protected identifier of up to four frames in the slave node with the configured NAD.
ld_assign_NAD	This call assigns the configured NAD (node diagnostic address) of all slave nodes that matches the initial_NAD, the supplier ID and the function ID.
ld_save_configuration	This call makes a save configuration request to a specific slave node with the given configured NAD or to all slave nodes if broadcast NAD is set.
ld_read_configuration	This call serializes the current configuration (configured NAD and PIDs) and copy it to the area (data pointer) provided by the application.
ld_set_configuration	The function configures the configured NAD and the PIDs according to the configuration provided.
IDENTIFICATION	
ld_read_by_id	The call requests the slave node selected with the configured NAD to return the property associated with the id parameter.
ld_read_by_id_callout	This callout is used when the master node transmits a read by identifier request with an identifier in the user defined area.
INITIALIZATION	
ld_init	This call reinitializes the raw or messaged-based layer on the interface.
RAW API	
ld_put_raw	The call queues the transmission of 8 bytes of data in one frame. The data is sent in the next suitable MasterReq frame.
ld_get_raw	The call copies the oldest received diagnostic frame data to the memory specified by data.
ld_raw_tx_status	The call returns the status of the raw frame transmission function.
ld_raw_rx_status	The call returns the status of the raw frame receive function.
MESSAGE-BASED API	
ld_send_message	The call packs the information specified by data and DataLength into one or multiple diagnostic frames.
ld_receive_message	The call prepares the LIN diagnostic module to receive one message and store it in the buffer pointed to by data.
ld_tx_status	The call returns the status of the last made call to ld_send_message.
ld_rx_status	The call returns the status of the last made call to ld_receive_message.

4.3.2 Data types

The LIN core defines the following types:

— l_bool 0 is false, and non-zero (>0) is true;

— l_ioctl_op implementation dependent;

— l_irqmask implementation dependent;

— l_u8 unsigned 8 bit integer;

— l_u16 unsigned 16 bit integer;

— l_signal_handle has character string type "signal name".

In order to gain efficiency, the majority of the functions are static functions (no parameters are needed, since one function exist per signal, per interface, etc.).

4.3.3 Driver and cluster management

4.3.3.1 l_sys_init

Table 2 defines the l_sys_init.

Table 2 — l_sys_init

Prototype	l_bool l_sys_init (void)
Applicability	Master and slave nodes.
Description	l_sys_init performs the initialization of the LIN core. The scope of the initialization is the physical node i.e. the complete node (see node composition definition in ISO 17987-2). The call to the l_sys_init is the first call a user uses in the LIN core before using any other API functions.
Return value	Zero if the initialization succeeded. Non-zero if the initialization failed.

4.3.4 Signal interaction

4.3.4.1 General

In all signal API calls below the sss is the name of the signal, e.g. l_u8_rd_enginespeed ().

4.3.4.2 Signal types

The signals are of three different types:

— l_bool for one bit signals; zero if false, non-zero otherwise;

— l_u8 for signals of the size 2 bits to 8 bits;

— l_u16 for signals of the size 9 bits to 16 bits.

4.3.4.3 Scalar signal read

Table 3 defines the scalar signal read.

Table 3 — Scalar signal read

Dynamic prototype	l_bool l_bool_rd (l_signal_handle sss); l_u8 l_u8_rd (l_signal_handle sss); l_u16 l_u16_rd (l_signal_handle sss);
Static prototype	l_bool l_bool_rd_sss (void); l_u8 l_u8_rd_sss (void); l_u16 l_u16_rd_sss (void);
Applicability	Master and slave nodes.
Description	Reads and returns the current value of the signal.
Reference	See ISO 17987-3:2016, 5.1.2.

4.3.4.4 Scalar signal write

Table 4 defines the scalar signal write.

Table 4 — Scalar signal write

Dynamic prototype	void l_bool_wr (l_signal_handle sss, l_bool v); void l_u8_wr (l_signal_handle sss, l_u8 v); void l_u16_wr (l_signal_handle sss, l_u16 v);
Static prototype	void l_bool_wr_sss (l_bool v); void l_u8_wr_sss (l_u8 v); void l_u16_wr_sss (l_u16 v);
Applicability	Master and slave nodes.
Description	Sets the current value of the signal to v.
Reference	See ISO 17987-3:2016, 5.1.2.

4.3.4.5 Byte array read

Table 5 defines the byte array read.

Table 5 — Byte array read

Dynamic prototype	void l_bytes_rd (l_signal_handle sss, l_u8 start, /* first byte to read from */ l_u8 count, /* number of bytes to read */ l_u8* const data); /* where data is written */
Static prototype	void l_bytes_rd_sss (l_u8 start, l_u8 count, l_u8* const data);
Applicability	Master and slave nodes.
Description	Reads and returns the current values of the selected bytes in the signal. The sum of start and count are never greater than the length of the byte array.
Example	Assume that a byte array is 6 bytes long, numbered 0 to 5. Reading byte 2 and 3 from this array indicates the parameter value start to be 2 (skipping byte 0 and 1) and count to be 2 (reading byte 2 and 3). In this case byte 2 is written to data [0] and byte 3 is written to data [1].
Reference	See ISO 17987-3:2016, 5.1.2.

4.3.4.6 Byte array write

Table 6 defines the byte array write.

Table 6 — Byte array write

Dynamic prototype	void l_bytes_wr (l_signal_handle sss, l_u8 start, /* first byte to write to */ l_u8 count, /* number of bytes to write */ const l_u8* const data); /* where data is read from */
Static prototype	void l_bytes_wr_sss (l_u8 start, l_u8 count, const l_u8* const data);
Applicability	Master and slave nodes.
Description	Sets the current value of the selected bytes in the signal specified by the name sss to the value specified. The sum of start and count are never greater than the length of the byte array, although the device driver does not choose to enforce this in runtime.
Example	Assume that a byte array is 7 bytes long, numbered 0 to 6. Writing byte 3 and 4 from this array indicates the parameter value start to be 3 (skipping byte 0, 1 and 2) and count to be 2 (writing byte 3 and 4). In this case byte 3 is read from data [0] and byte 4 is read from data [1].
Reference	See ISO 17987-3:2016, 5.1.2.

4.3.5 Notification

Flags are local objects in a node and they are used to synchronize the application program with the LIN core. The flags are automatically set by the LIN core and can only be tested or cleared by the application program. Flags are attached to all types of frames. A flag is set when the frame/signal is considered to be transmitted respectively received, see reception and transmission in ISO 17987-3.

Three types of flags can be created:

— a flag that is attached to a signal,

— a flag that is attached to a frame, and

— a flag that is attached to a signal in a particular frame. This is used when a signal is packed into multiple frames.

All three listed flag types above are applicable on both transmitted and received signals/frames.

4.3.5.1 l_flg_tst

Table 7 defines the I_flg_tst.

Table 7 — I_flg_tst

Dynamic prototype	l_bool l_flg_tst (l_flag_handle fff)
Static prototype	l_bool l_flg_tst_fff (void); Where fff is the name of the flag, e.g. l_flg_tst_RxEngineSpeed ().
Applicability	Master and slave nodes.
Description	Returns a C boolean indicating the current state of the flag specified by the name fff, i.e. returns zero if the flag is cleared, non-zero otherwise.
Example	A flag named tx confirmation is attached to a published signal valve position stored in the IO_1 frame. The static implementation of the l_flg_tst is: l_bool l_flg_tst_txconfirmation (void); The flag is set when the IO_1 frame (containing the signal value position) is successfully transmitted from the node.
Reference	No reference, flags are API specific and not described anywhere else.

4.3.5.2 l_flg_clr

Table 8 defines the l_flg_clr.

Table 8 — l_flg_clr

Dynamic prototype	void l_flg_clr (l_flag_handle fff);
Static prototype	void l_flg_clr_fff (void); Where fff is the name of the flag, e.g. l_flg_clr_RxEngineSpeed ().
Applicability	Master and slave nodes.
Description	Sets the current value of the flag specified by the name fff to zero.
Reference	No reference, flags are API specific and not described anywhere else.

4.3.6 Schedule management

4.3.6.1 l_sch_tick

Table 9 defines the l_sch_tick.

Table 9 — l_sch_tick

Dynamic prototype	l_u16 l_sch_tick (l_ifc_handle iii);
Static prototype	l_u16 l_sch_tick_iii (void); where iii is the name of the interface, e.g. l_sch_tick_MyLinIfc ().
Applicability	Master nodes only.
Description	The l_sch_tick function provides the LIN driver a time base for the scheduler. When a frame becomes due, its transmission is initiated. When the end of the current schedule is reached, l_sch_tick starts again at the beginning of the schedule. The l_sch_tick is called periodically and individually for each interface within the node. The period is the time base, see ISO 17987-3:2016, 5.3, set in the LDF, see ISO 17987-3:2016, 12.3.4.2. The period of the l_sch_tick call effectively sets the time base tick, see ISO 17987-3:2016, 5.3. Therefore, it is essential that the time base period is uphold with minimum jitter. The call to l_sch_tick does not only start the transition of the next frame due, it also updates the signal values for those signals received since the previous call to l_sch_tick, see ISO 17987-3:2016, 5.1.5.
Return value	Zero, if the next call of l_sch_tick does not start transmission of a frame. Non-zero, if the next call of l_sch_tick starts the transmission of the frame in the next schedule table entry. The return value in this case is the next schedule table entry's number (counted from the beginning of the schedule table) in the schedule table. The return value is in range 1 to N if the schedule table has N entries.
Reference	See ISO 17987-3:2016, 5.3.

4.3.6.2 l_sch_set

Table 10 defines the l_sch_set.

Table 10 — l_sch_set

Dynamic prototype	void l_sch_set (l_ifc_handle iii, l_schedule_handle schedule, l_u16 entry);
Static prototype	void l_sch_set_iii (l_schedule_handle schedule, l_u16 entry); where iii is the name of the interface, e.g. l_sch_set_MyLinIfc (MySchedule1, 0).
Applicability	Master node only.
Description	Sets up the next schedule to be followed by the l_sch_tick function for a certain interface iii. The new schedule is activated as soon as the current schedule reaches its next schedule entry point. The extension “iii” is the interface name. It is optional and the intention is to solve naming conflicts when the node is a master on more than one LIN cluster. The entry defines the starting entry point in the new schedule table. The value is in the range 0 to N if the schedule table has N entries, and if entry is 0 or 1 the new schedule table is started from the beginning. A predefined schedule table, L_NULL_SCHEDULE, exists and is used to stop all transfers on the LIN cluster.
Example	A possible use of the entry value is in combination with the l_sch_tick return value to temporarily interrupt one schedule with another schedule table and still be able to switch back to the interrupted schedule table at the point where this was interrupted.
Reference	See ISO 17987-3:2016, 5.3.

4.3.7 Interface management

4.3.7.1 General

Interface management calls manage the specific interfaces (the logical channels to the bus). Each interface is identified uniquely by its interface name, denoted by the iii extension for each API call. How to set the interface name (iii) is not in the scope of this document.

4.3.7.2 l_ifc_init

Table 11 defines the l_ifc_init.

Table 11 — l_ifc_init

Dynamic prototype	l_bool l_ifc_init (l_ifc_handle iii)
Static prototype	l_bool_ifc_init_iii (void); Where iii is the name of the interface, e.g. l_ifc_init_MyLinIfc ().
Applicability	Master and slave nodes.
Description	l_ifc_init initializes the controller specified by the name iii, i.e. sets up internal functions such as the baud rate. The default schedule set in a master node by the l_ifc_init call is the L_NULL_SCHEDULE where no frames are sent and received. This is the first call a user performs before using any other interface related LIN API functions. The function returns zero if the initialisation was successful and non-zero if failed.
Reference	A general description of the interface concept is found in concept of operation in ISO 17987-3.

4.3.7.3 l_ifc_goto_sleep

Table 12 defines the l_ifc_goto_sleep.

Table 12 — l_ifc_goto_sleep

Dynamic prototype	void l_ifc_goto_sleep (l_ifc_handle iii)
Static prototype	void l_ifc_goto_sleep_iii (void); Where iii is the name of the interface, e.g. l_ifc_goto_sleep_MyLinIfc ().
Applicability	Master node only.
Description	This call requests slave nodes on the cluster connected to the interface to enter bus sleep mode by issuing one go to sleep command, see ISO 17987-2:2016, 5.4. The go to sleep command is scheduled latest when the next schedule entry is due. The l_ifc_goto_sleep does not affect the power mode. It is up to the application to do this. If the go to sleep command was successfully transmitted on the cluster the go to sleep bit is set in the status register, see ISO 17987-2:2016, 5.4.
Reference	See ISO 17987-2:2016, 5.4.

4.3.7.4 l_ifc_wake_up

Table 13 defines the l_ifc_wake_up.

Table 13 — l_ifc_wake_up

Dynamic prototype	void l_ifc_wake_up (l_ifc_handle iii)
Static prototype	void l_ifc_wake_up_iii (void); where iii is the name of the interface, e.g. l_ifc_wake_up_MyLinIfc ().
Applicability	Master and slave nodes.
Description	The function transmits one wake up signal. The wake up signal is transmitted directly when this function is called. It is the responsibility of the application to retransmit the wake up signal according to the wake up sequence defined in ISO 17987-2
Reference	See ISO 17987-2:2016, 5.3.

4.3.7.5 l_ifc_ioctl

Table 14 defines the l_ifc_ioctl.

Table 14 — l_ifc_ioctl

Dynamic prototype	l_u16 l_ifc_ioctl (l_ifc_handle iii, l_ioctl_op op, void* pv)
Static prototype	l_u16 l_ifc_ioctl_iii (l_ioctl_op op, void* pv); where iii is the name of the interface, e.g. l_ifc_ioctl_MyLinIfc (MyOp, &MyPars).
Applicability	Master and slave nodes.
Description	This function controls functionality that is not covered by the other API calls. It is used for protocol specific parameters or hardware specific functionality. Example of such functionality can be to switch on/off the wake up signal detection. The iii is the name of the interface to which the operation defined in op ise applied. The pointer pv points to an optional parameter that is provided to the function. Exactly which operations that are supported is implementation dependent.
Reference	No reference, the behaviour is API specific and not described anywhere else.

4.3.7.6 l_ifc_rx

Table 15 defines the l_ifc_rx.

Table 15 — l_ifc_rx

Dynamic prototype	void l_ifc_rx (l_ifc_handle iii)
Static prototype	void l_ifc_rx_iii (void); where iii is the name of the interface, e.g. l_ifc_rx_MyLinIfc ().
Applicability	Master and slave nodes.
Description	The application program is responsible for binding the interrupt and for setting the correct interface handle (if interrupt is used). For UART based implementations, it is called from a user-defined interrupt handler triggered by a UART when it receives one character of data. In this case, the function performs necessary operations on the UART control registers. For more complex LIN hardware, it is used to indicate the reception of a complete header or frame.
Reference	No reference, the behaviour is API specific and not described anywhere else.

4.3.7.7 l_ifc_tx

Table 16 defines the l_ifc_tx.

Table 16 — l_ifc_tx

Dynamic prototype	void l_ifc_tx (l_ifc_handle iii)
Static prototype	void l_ifc_tx_iii (void); where iii is the name of the interface, e.g. l_ifc_tx_MyLinIfc ().
Applicability	Master and slave nodes.
Description	The application program is responsible for binding the interrupt and for setting the correct interface handle (if interrupt is used). For UART based implementations, it is called from a user-defined interrupt handler triggered by a UART when it has transmitted one character of data. In this case the function performs necessary operations on the UART control registers. For more complex LIN hardware, it is used to indicate the transmission of a complete frame.
Reference	No reference, the behaviour is API specific and not described anywhere else.

4.3.7.8 l_ifc_aux

Table 17 defines the l_ifc_aux.

Table 17 — l_ifc_aux

Dynamic prototype	void l_ifc_aux (l_ifc_handle iii)
Static prototype	void l_ifc_aux_iii (void); Where iii is the name of the interface, e.g. l_ifc_aux_MyLinIfc ().
Applicability	Master and slave nodes.
Description	This function is used in the slave nodes to synchronize to the break field/sync byte field sequence transmitted by the master node on the interface specified by iii. It is called, for example, from a user-defined interrupt handler raised upon an edge detection on a hardware pin connected to the interface iii. l_ifc_aux is only used in a slave node. This function is strongly hardware connected and the exact implementation and usage is implementation dependent. This function might even be empty in cases where the break field/sync byte field sequence detection is implemented in the l_ifc_rx function.
Reference	No reference, the behaviour is API specific and not described anywhere else.

4.3.7.9 l_ifc_read_status

Table 18 defines the l_ifc_read_status.

Table 18 — l_ifc_read_status

Dynamic prototype	l_u16 l_ifc_read_status (l_ifc_handle iii)
Static prototype	l_u16 l_ifc_read_status_iii (void); where iii is the name of the interface, e.g. l_ifc_read_status_MyLinIfc ().
Applicability	Master and slave nodes. The behaviour is different for master and slave nodes, see description below.
Description	This function returns the status of the previous communication. The call returns the status word (16 bit value), as shown in Table 19.
Reference	See ISO 17987-3:2016, 5.5.

Table 19 defines the return value of l_ifc_read_status (bit 15 is MSB, bit 0 is LSB).

Table 19 — Return value of l_ifc_read_status (bit 15 is MSB, bit 0 is LSB).

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Last frame PID								0	Save configuration	Event triggered frame collision	Bus activity	Go to sleep	Overrun	Successful transfer	Error in response

The status word is only set based on a frame transmitted or received by the node (except bus activity).

The call is a read-reset call; meaning that after the call has returned, the status word is set to 0.

In the master node, the status word is updated in the l_sch_tick function. In the slave node, the status word is updated latest when the next frame is started.

Error in response is set if a frame error is detected in the frame response, e.g. checksum error, framing error, etc. An error in the header results in the header not being recognized and thus, the frame is ignored. It is not set if there was no response on a received frame. Also, it is not be set if there is an error in the response collision) of an event triggered frame.

Successful transfer is set if a frame has been transmitted/received without an error.

Overrun is set if two or more frames are processed since the previous call to l_ifc_read_status. If this is the case, error in response and successful transfer represent logical ORed values for all processed frames.

Go to sleep is set in a slave node if a go to sleep command has been received and set in a master node when the go to sleep command is successfully transmitted on the bus. After receiving the go to sleep command the power mode is not affected. This is done in the application.

Bus activity is set if the node has detected bus activity on the bus. For the definition of bus activity, see go to sleep in ISO 17987-2. A slave node enters bus sleep mode after a period of bus inactivity on the bus, see go to sleep in ISO 17987-2. This can be implemented by the application monitoring the bus activity. Note the difference between bus activity and bus inactivity.

Event triggered frame collision is set as long the collision resolving schedule is executed. The intention is to use it in parallel with the return value from l_sch_tick. In the slave, this bit is always 0 (zero). If the master node application switches schedule table during the collision is resolved, the event triggered frame collision flag is set to 0 (zero). See example below how this flag is set.

Save configuration is set when the save configuration request has been successfully received, see ISO 17987-3:2016, 6.3.5. It is set only in the slave node, in the master node it is always 0 (zero).

Last frame PID is the PID last detected on the bus and processed in the node. If over-run is set one or more values of last frame PID are lost, only the latest value is maintained. It is set simultaneously with successful transfer or error in response.

The combination of the two status bits successful transfer and error in response is interpreted according to Table 20.

Table 20 — Node internal error interpretation

Error in response	Successful transfer	Interpretation
0	0	No communication or no response
1	1	Intermittent communication (some successful transfers and some failed)
0	1	Full communication
1	0	Erroneous communication (only failed transfers)

It is the responsibility of the node application to process the individual status reports (see ISO 17987-3).

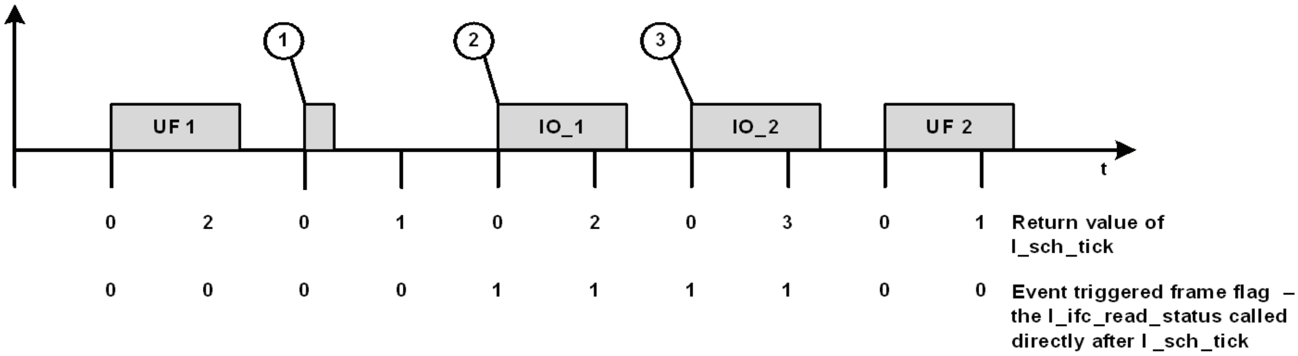
EXAMPLE 1 The l_ifc_read_status is designed to allow reading at a much lower frequency than the frame slot frequency, e.g. once every 50 frame slots. In this case, the last frame PID has little use. Overrun is then used as a check that the traffic is running as intended, i.e. is always be set. It is, however, also possible to call l_ifc_read_status every frame slot and get a much better error statistics, you can see the protected identifier of the failing transfers and by knowing the topology, it is possible to draw better conclusion of faulty nodes. This is maybe most useful in the master node, but is also possible in any slave node.

EXAMPLE 2 This example shows how the event triggered flag behaves in case of a collision resolving. The normal schedule table is depicted in Table 21.

Table 21 — Event triggered frame example schedule table

Frame	Delay	Frame type
UF1	10 ms	unconditional
IO_check	10 ms	event triggered
UF2	10 ms	unconditional

The IO_1 and IO_2 unconditional frames are associated with IO_check. The collision solving schedule table contains the unconditional frames IO_1 and IO_2 (with delays set to 10 ms). The collision is handled as shown in Figure 1. The time base in this example is set to 5 ms.



Key

1 master node transmits header of IO_check but both slave nodes responded, i.e. a collision occurs

2 master node switches automatically to the collision solving schedule table

3 switches automatically back to the normal schedule table

Figure 1 — Event triggered frame collision solving example

4.3.8 User provided call outs

The application provides a pair of functions, which is called (implementation dependent) from within the LIN module in order to disable LIN communication interrupts before certain internal operations and to restore the previous state after such operations. These functions can, for example, be used in the l_sch_tick function. The application itself also makes use of these functions.

4.3.8.1 l_sys_irq_disable

Table 22 defines the l_sys_irq_disable.

Table 22 — l_sys_irq_disable

Prototype	l_irqmask l_sys_irq_disable (void)
Applicability	Master and slave nodes.
Description	The user implementation of this function achieves a state in which no interrupts from the LIN communication can occur.
Reference	No reference, the behaviour is API specific and not described anywhere else.

4.3.8.2 l_sys_irq_restore

Table 23 defines the l_sys_irq_restore.

Table 23 — l_sys_irq_restore

Prototype	void l_sys_irq_restore (l_irqmask previous)
Applicability	Master and slave nodes.
Description	The user implementation of this function restores the interrupt level identified by the provided l_irqmask previous.
Reference	No reference, the behaviour is API specific and not described anywhere else.

4.4 Node configuration and identification

4.4.1 Overview

The node configuration and diagnostic API has a set of functions all based on the idea to give the API a separate name space, in order to minimize the risk of conflicts with existing software. All functions and types have the prefix "ld_" (lowercase "LD" followed by an "underscore").

For operation of the node configuration, the master request frame and slave response frame are scheduled. If the master node does not regard the responses of the requests only the master request frame is contained in the schedule table.

4.4.2 Node configuration

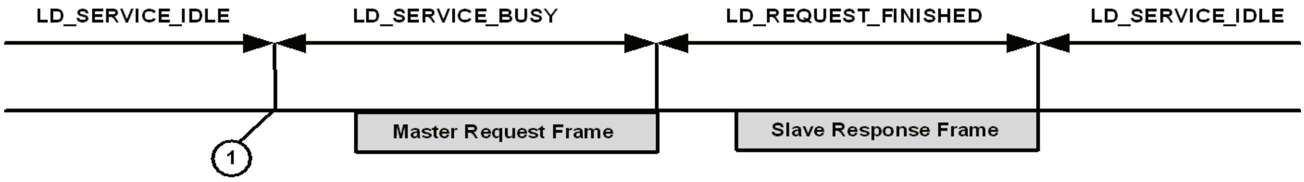
4.4.2.1 ld_is_ready

Table 24 defines the ld_is_ready.

Table 24 — ld_is_ready

Prototype	l_u8 ld_is_ready (l_ifc_handle iii)
Applicability	Master node only.
Description	This call returns the status of the last requested configuration service.
Return value	LD_SERVICE_BUSY: Service is ongoing. LD_REQUEST_FINISHED: The configuration request has been completed. This is an intermediate status between the configuration request and configuration response. LD_SERVICE_IDLE: The configuration request/response combination has been completed, i.e. the response is valid and is analyzed. Also, this value is returned if no request has yet been called. LD_SERVICE_ERROR The configuration request or response experienced an error. Error here means error on the bus, and not a negative configuration response from the slave node.
Reference	No reference, the behaviour is API specific and not described anywhere else.

Figure 2 shows the situation where a successful configuration request and configuration response is made. Note that the state change after the master request frame and slave response frame are finished is delayed up to one time base.



Key

1 configuration service called

Figure 2 — Successful configuration request and configuration response

4.4.2.2 ld_check_response

Table 25 defines the ld_check_response.

Table 25 — ld_check_response

Prototype	void ld_check_response (l_ifc_handle iii, l_u8* const RSID, l_u8* const error_code)
Applicability	Master node only.
Description	This call returns the result of the last node configuration service, in the parameters RSID and error_code. A value in RSID is always returned but not always in the error_code. Default values for RSID and error_code is 0 (zero).
Reference	No reference, the behaviour is API specific and not described anywhere else.

4.4.2.3 ld_assign_frame_id_range

Table 26 defines the ld_assign_frame_id_range.

Table 26 — ld_assign_frame_id_range

Prototype	void ld_assign_frame_id_range (l_ifc_handle iii, l_u8 NAD, l_u8 start_index, const l_u8* const PIDs)
Applicability	Master node only.
Description	This call assigns the protected identifier of up to four frames in the slave node with the addressed configured NAD. The PIDs parameter is four bytes long, each byte contains a PID, do not care or unassign value.
Reference	See ISO 17987-3:2016, 6.3.6.2.

4.4.2.4 ld_assign_NAD

Table 27 defines the ld_assign_NAD.

Table 27 — ld_assign_NAD

Prototype	void ld_assign_NAD (l_ifc_handle iii, l_u8 initial_NAD, l_u16 supplier_id, l_u16 function_id, l_u8 configured_NAD)
Applicability	Master node only.
Description	This call assigns the configured_NAD to the slave nodes that matches the initial_NAD, the supplier_id and the function_id.
Reference	See the definition of the service assign NAD, see ISO 17987-3.

4.4.2.5 ld_save_configuration

Table 28 defines the ld_save_configuration.

Table 28 — ld_save_configuration

Prototype	void ld_save_configuration (l_ifc_handle iii, l_u8 NAD)
Applicability	Master node only.
Description	This call makes a save configuration request to a specific slave node with the given configured NAD, or to all slave nodes if broadcast NAD is set.
Reference	See the definition of the save configuration service in ISO 17987-3. API call l_ifc_read_status see 4.3.7.9 and example in 4.6.

4.4.2.6 ld_read_configuration

Table 29 defines the ld_read_configuration.

Table 29 — ld_read_configuration

Prototype	l_u8 ld_read_configuration (l_ifc_handle iii, l_u8* const data, l_u8* const length)
Applicability	Slave node only.
Description	This function does not transport anything on the bus. This call serializes the current configuration and copy it to the area (data pointer) provided by the application. The intention is to call this function when the save configuration request flag is set in the status register, see 4.3.7.9. After the call is finished the application is responsible to store the data in appropriate memory. The caller reserves bytes in the data area equal to length, before calling this function. The function sets the length parameter to the actual size of the configuration. In case the data area is too short the function returns with no action. In case the NAD has not been set by a previous call to ld_set_configuration or the master node has used the configuration services, the returned configured NAD still has the value of the initial NAD. The data contains the configured NAD and the PIDs and occupies one byte each. The structure of the data is: configured NAD and then all PIDs for the frames. The order of the PIDs is the same as the frame list in the LDF and the frame definition in the NCF, both in ISO 17987-2.
Return value	LD_READ_OK: If the service was successful. LD_LENGTH_TOO_SHORT: If the configuration size is greater than the length. It means that the data area does not contain a valid configuration.
Reference	See the definition of the save configuration service in ISO 17987-3. Function l_ifc_read_status see 4.3.7.9 and example in 4.6.

4.4.2.7 ld_set_configuration

Table 30 defines the ld_set_configuration.

Table 30 — ld_set_configuration

Prototype	l_u8 ld_set_configuration (l_ifc_handle iii, const l_u8* const data, l_u16 length)
Applicability	Slave node only.
Description	This call does not transport anything on the bus. The function configures the NAD and the PIDs according to the configuration given by data. The intended usage is to restore a saved configuration or set an initial configuration (e.g. coded by I/O pins). The function is called after calling l_ifc_init. The caller sets the size of the data area before calling the function. The data contains the configured NAD and the PIDs and occupies one byte each. The structure of the data is NAD and then all PIDs for the frames. The order of the PIDs is the same as the frame list in the LDF and the frame definition in the NCF, both in ISO 17987-2.
Return value	LD_SET_OK: If the service was successful. LD_LENGTH_NOT_CORRECT: If the size of the configuration is not equal to the given length. LD_DATA_ERROR: The set of configuration could not be made.
Reference	See the definition of the save configuration service in ISO 17987-3. Function l_ifc_read_status, see 4.3.7.9 and example in 4.6.

4.4.3 Identification

4.4.3.1 ld_read_by_id

Table 31 defines the ld_read_by_id.

Table 31 — ld_read_by_id

Prototype	void ld_read_by_id (l_ifc_handle iii, l_u8 NAD, l_u16 supplier_id, l_u16 function_id, l_u8 id, l_u8* const data);
Applicability	Master node only.
Description	The call requests the slave node selected with the NAD to return the property associated with the id parameter, see ISO 17987-3:2016, Table 20 for interpretation of the id. When the next call to ld_is_ready returns LD_SERVICE_IDLE (after the ld_read_by_id is called), the RAM area specified by data contains between one and five bytes data according to the request. The result is returned in a big endian style. It is up to little endian CPUs to swap the bytes, not the LIN diagnostic driver. The reason for using big endian data is to simplify message routing to a (e.g. CAN) backbone network.
Reference	See definition of the ReadByIdentifier service in ISO 17987-3.

4.4.3.2 ld_read_by_id_callout

Table 32 defines the ld_read_by_id_callout.

Table 32 — ld_read_by_id_callout

Prototype	l_u8 ld_read_by_id_callout (l_ifc_handle iii, l_u8 id, l_u8* data)
Applicability	This callout is optional and is available in slave node only. In case the user defined read by identifier request is used, the slave node application implements this call-out.
Description	This callout is used when the master node transmits a ReadByIdentifier request with an identifier in the user defined area. The slave node application is called from the driver when such request is received. The id parameter is the identifier in the user defined area (32 to 63), see ISO 17987-3:2016, Table 18 from the ReadByIdentifier configuration request. The data pointer points to a data area with 5 bytes. This area is used by the application to set up the positive response, see the user defined area in ISO 17987-3:2016, Table 19.
Return value	LD_NEGATIVE_RESPONSE: The slave node responds with a negative response as defined in ISO 17987-3:2016, Table 20. In this case, the data area is not considered. LD_POSTIVE_RESPONSE: The slave node sets up a positive response using the data provided by the application. LD_NO_RESPONSE: The slave node does not answer.
Reference	See ISO 17987-3:2016, Clause 6.

4.5 Transport layer

4.5.1 Overview

The LIN transport layer API has a set of functions all based on the idea to give the API a separate name space, in order to minimize the risk of conflicts with existing software. All functions and types have the prefix “ld_” (lowercase “LD” followed by an “underscore”).

Use of the LIN diagnostic transport layer API demands knowledge of the underlying protocol. The relevant information can be found in ISO 17987-2. LIN diagnostic transport layer is intended to transport diagnostic requests/responses between a test equipment on a (e.g. CAN) backbone network to LIN slave nodes via the master node.

4.5.2 Raw- and messaged-based API

Since ISO 15765-2[4] PDUs on CAN are quite similar to LIN diagnostic frames, a raw API is provided. The raw API is frame-/PDU-based and it is up to the application to manage the PCI information. The idea of the raw API is to interface to the CAN transport layer. With small efforts and resources the raw API can be used to gateway diagnostic requests/responds between CAN and LIN. A prerequisite for the raw API is that the frame format on CAN is equivalent to LIN where the addressing information is stored in the first byte.

The messaged-based API is message based. The application provides a pointer to a message buffer. When the transfer commences, the LIN driver does the packing/unpacking, i.e. act as a transport layer. Typically, this is useful in slave nodes since they do not gateway the messages but parse them.

Both raw API and the messaged-based API use the same structure of the diagnostic frames, i.e. NAD, PCI, SID, etc.

4.5.3 Initialization

4.5.3.1 ld_init

Table 33 defines the ld_init.

Table 33 — ld_init

Prototype	void ld_init (l_ifc_handle iii)
Applicability	Master and slave nodes.
Description	This call (re)initializes the raw or messaged-based layers on the interface iii. All transport layer buffers are initialized. If there is an ongoing diagnostic frame transporting a messaged-based or raw message on the bus, it is not aborted.
Reference	No reference, the behaviour is API specific and not described anywhere else.

4.5.4 Raw API

4.5.5 Overview

The raw API is operating on PDU level and it is typically used to gateway PDUs between CAN and LIN. Usually, a FIFO is used to buffer PDUs in order to handle the different bus speeds.

4.5.5.1 ld_put_raw

Table 34 defines the ld_put_raw.

Table 34 — ld_put_raw

Prototype	void ld_put_raw (l_ifc_handle iii, const l_u8* const data)
Applicability	Master nodes.
Description	The call queues the transmission of 8 bytes of data in one frame. The data is sent in the next suitable frame (MasterReq frame). The data area is copied in the call, the pointer is not memorized. If no more queue resources are available, the data is jettisoned and the appropriate error status is set.
Reference	The raw and messaged-based is not differentiated outside the API. A general description of the transport layer can be found in ISO 17987-2.

4.5.5.2 ld_get_raw

Table 35 defines the ld_get_raw.

Table 35 — ld_get_raw

Prototype	void ld_get_raw (l_ifc_handle iii, l_u8* const data)
Applicability	Master nodes.
Description	The call copies the oldest received diagnostic frame data to the memory specified by data. The data returned is received from SlaveResp frame. If the receive queue is empty no data is copied.
Reference	The raw and messaged-based is not differentiated outside the API. A general description of the transport layer can be found in ISO 17987-2.

4.5.5.3 ld_raw_tx_status

Table 36 defines the ld_raw_tx_status.

Table 36 — ld_raw_tx_status

Prototype	l_u8 ld_raw_tx_status (l_ifc_handle iii)
Applicability	Master nodes.
Description	The call returns the status of the raw frame transmission function:
Return values	LD_QUEUE_EMPTY: The transmit queue is empty. In case previous calls to ld_put_raw, all frames in the queue have been transmitted. LD_QUEUE_AVAILABLE: The transmit queue contains entries, but is not full. LD_QUEUE_FULL: The transmit queue is full and cannot accept further frames. LD_TRANSMIT_ERROR: LIN protocol errors occurred during the transfer; initialize and redo the transfer.
Reference	The raw and messaged-based is not differentiated outside the API. A general description of the transport layer can be found in ISO 17987-2.

4.5.5.4 ld_raw_rx_status

Table 37 defines the ld_raw_rx_status.

Table 37 — ld_raw_rx_status

Prototype	l_u8 ld_raw_rx_status (l_ifc_handle iii)
Applicability	Master nodes.
Description	The call returns the status of the raw frame receive function:
Return values	LD_NO_DATA: The receive queue is empty. LD_DATA_AVAILABLE: The receive queue contains data that can be read. LD_RECEIVE_ERROR: LIN protocol errors occurred during the transfer; initialize and redo the transfer.
Reference	The raw and messaged-based is not differentiated outside the API. A general description of the transport layer can be found in ISO 17987-2.

4.5.6 Messaged-based API

4.5.6.1 Overview

Messaged-based processing of diagnostic messages manages one complete message at a time.

4.5.6.2 ld_send_message

Table 38 defines the ld_send_message.

Table 38 — ld_send_message

Prototype	void ld_send_message (l_ifc_handle iii, l_u16 DataLength, l_u8 NAD, const l_u8* const data)
Applicability	Master and slave nodes.
Description	The call packs the information specified by data and DataLength into one or multiple diagnostic frames. If the call is made in a master node application, the frames are transmitted to the slave node with the address NAD. If the call is made in a slave node application, the frames are transmitted to the master node with the address NAD. The parameter NAD is not used in slave nodes. The value of the SID (or RSID) is the first byte in the data area. DataLength is in the range of 1 to 4095 bytes. The DataLength also includes the SID (or RSID) value, i.e. message length plus one. The call is asynchronous, i.e. not suspended until the message has been sent, and the buffer does not be changed by the application as long as calls to ld_tx_status returns LD_IN_PROGRESS. The data is transmitted in suitable frames (master request frame for master nodes and slave response frame for slave nodes). If there is a message in progress, the call returns with no action.
Reference	The raw and messaged-based is not differentiated outside the API. A general description of the transport layer can be found in ISO 17987-2.

4.5.6.3 ld_receive_message

Table 39 defines the ld_receive_message.

Table 39 — ld_receive_message

Prototype	void ld_receive_message (l_ifc_handle iii, l_u16* const DataLength, l_u8* const NAD, l_u8* const data)
Applicability	Master and slave nodes.
Description	The call prepares the LIN diagnostic module to receive one message and store it in the buffer pointed to by data. At the call, DataLength specifies the maximum length allowed. When the reception has completed, DataLength is changed to the actual length and NAD to the NAD in the message. SID (or RSID) is the first byte in the data area. DataLength is in the range of 1 to 4 095 bytes, but never more than the value originally set in the call. SID (or RSID) is included in the DataLength. The parameter NAD is not used in slave nodes. The call is asynchronous, i.e. not suspended until the message has been received, and the buffer is not changed by the application as long as calls to ld_rx_status returns LD_IN_PROGRESS. If the call is made after the message transmission has commenced on the bus (i.e. the SF or FF is already transmitted), this message is not received. Instead the function waits until next message commence. The data is received from the succeeding suitable frames (master request frame for slave nodes and slave response frame for master nodes). The application monitors the ld_rx_status and does not call this function until the status is LD_COMPLETED. Otherwise, this function returns inconsistent data in the parameters.
Reference	The raw and messaged-based is not differentiated outside the API. A general description of the transport layer can be found in ISO 17987-2.

4.5.6.4 ld_tx_status

Table 40 defines the ld_tx_status.

Table 40 — ld_tx_status

Prototype	l_u8 ld_tx_status (l_ifc_handle iii)
Applicability	Master and slave nodes.
Description	The call returns the status of the last made call to ld_send_message. The following values can be returned.
Return values	LD_IN_PROGRESS: The transmission is not yet completed. LD_COMPLETED: The transmission has completed successfully (and you can issue a new ld_send_message call). This value is also returned after initialization of the transport layer. LD_FAILED: The transmission ended in an error. The data was only partially sent. The transport layer is reinitialized before processing further messages. To find out why a transmission has failed, check the status management function l_ifc_read_status, see 4.3.7.9. LD_N_AS_TIMEOUT: The transmission failed because of a N_As timeout, see ISO 17987-2.
Reference	The raw and messaged-based is not differentiated outside the API. A general description of the transport layer can be found in ISO 17987-2.

4.5.6.5 ld_rx_status

Table 41 defines the ld_rx_status.

Table 41 — ld_rx_status

Prototype	l_u8 ld_rx_status (l_ifc_handle iii)
Applicability	Master and slave nodes.
Description	The call returns the status of the last made call to ld_receive_message. The following values can be returned.
Return values	LD_IN_PROGRESS: The reception is not yet completed. LD_COMPLETED: The reception has completed successfully and all information (DataLength, NAD, data) is available. (You can also issue a new ld_receive_message call). This value is also returned after initialization of the transport layer. LD_FAILED: The reception ended in an error. The data was only partially received and is not trusted. Initialize before processing further transport layer messages. To find out why a reception has failed, check the status management function l_ifc_read_status, see 4.3.7.9. LD_N_CR_TIMEOUT: The reception failed because of an N_Cr timeout, see ISO 17987-2. LD_WRONG_SN: The reception failed because of an unexpected sequence number.
Reference	The raw and messaged-based is not differentiated outside the API. A general description of the transport layer can be found in ISO 17987-2.

4.6 Examples

4.6.1 Overview

Two examples are included to show how the API can be used:

— master node example, and

— slave node example.

The examples are not complete; there are functions that are not implemented.

4.6.2 Master node example

```
/*********************************************************************
* Description : Example code for using the LIN API in a LIN master node
*               The static LIN API is used
*********************************************************************/

#include <lin.h>
#define INT_ENABLE_LEVEL 1

/*********************************************************************
* Procedure      : l_sys_irq_restore
* Description    : Restores the interrupt mask to the one before the call
*                  to l_sys_irq_disable was made
* In parameters  : previous - the old interrupt level
* Out parameters : None
* Return value   : void
```

```
*******************************************************************/

void l_sys_irq_restore (l_irqmask previous)
{
  /* Set interrupt level to previous */
} /* l_sys_irq_restore */

/*******************************************************************
* Procedure      : l_sys_irq_disable
* Description    : Disable the UART interrupts of the controller and
*                  return the interrupt level to be able to restore it
*                  later
* In parameters  : None
* Out parameters : None
* Return value   : The interrupt level before disable
*******************************************************************/

l_irqmask l_sys_irq_disable (void)
{
  l_irqmask interrupt_level;
  /* Store the interrupt level and then disable UART interrupts */
  return interrupt_level;
} /* l_sys_irq_disable */

/*******************************************************************
* Interrupt      : lin_char_rx_handler
* Description    : UART receive character interrupt handler for the
*                  interface i1
* In parameters  : None
* Out parameters : None
* Return value   : void
*******************************************************************/

void __INTERRUPT /* Compiler intrinsic */ lin_char_rx_handler (void)
{
  /* Just call the LIN API provided function to do the actual work */
  l_ifc_rx_i1 ();
} /* lin_char_rx_handler */

/*******************************************************************
* Procedure      : main
* Description    : Main entry of application
* In parameters  : None
* Out parameters : None
* Return value   : function never returns
*******************************************************************/

int main (void)
{
  /* Initialize the LIN interface */
  if (l_sys_init ()) {
    /* The init of the LIN software failed - call error routine */
  }
  /* Initialize the interface */
  if (l_ifc_init_i1 ()) {
    /* Initialization of the LIN interface failed - call error routine */
  }
  /* Now is the first time the LIN interrupts can be enabled */
  l_sys_irq_restore (INT_ENABLE_LEVEL);
  /* Set the normal schedule */
  l_sch_set_i1 (Normal_Schedule, 0);
  /* Start the OS */
  start_OS ();
  /* return code */
  return 1;
} /* main */

/*******************************************************************
* Procedure      : main_application_10ms
* Description    : Main 10 ms task of the application
* In parameters  : None
```



```
* Out parameters : None
* Return value   : void */
*********************************************************************/

void main_application_10ms (void)
{
  /* In/output of signals. Call it first in the task to minimize jitter */
  (void) l_sch_tick_i1();
  /* Do some application specific stuff... */
  /* Just a small example of frame receive check and signal writing */
  if (l_flg_tst_RxInternalLightsSwitch ())
  {
    l_flg_clr_RxInternalLightsSwitch ();
    /* signal reading and writing */
    l_u8_wr_InternalLightsRequest (l_u8_rd_InternalLightsSwitch());
  }
} /* main_application_10ms */
```

4.6.3 Slave node example

The following example shows how a simple application in a slave is made. Special focus is made on the node configuration.

```
/*********************************************************************
* Description  : Example code for using the LIN API in a LIN slave node.
*                The static LIN API is used (for the core API)
*********************************************************************/

#include "lin.h"
#define INT_ENABLE_LEVEL 1

/*********************************************************************

* Interrupt      : lin_char_rx_handler
* Description    : UART receive character interrupt handler for the
*                  interface i1
* In parameters  : None
* Out parameters : None
* Return value   : void
*********************************************************************/

void __INTERRUPT /* Compiler intrinsic */ lin_char_rx_handler (void)
{
  /* Just call the LIN API provided function to do the actual work */
  l_ifc_rx_i1 ();
} /* lin_char_rx_handler */

/*********************************************************************
* Procedure      : main_task
* Description    : Main task covering LIN functionalities
* In parameters  : None
* Out parameters : None
* Return value   : void */
*********************************************************************/

void main_task (void)
{
  /* Do some application specific stuff... */
  /* poll frame received status */
  if (l_flg_tst_InternalLightsRequest_flag ())
  {
    /* clear the flag */
    l_flg_clr_InternalLightsRequest_flag ();
    /* Just a small example of signal and flag handling */
    if (l_u8_rd_InternalLightsSwitch () == 1) {
      /* turn on lights */
    }
  }
} /* main_task */

/*********************************************************************
```

```
* Procedure      : main
* Description    : Main entry of application
* In parameters  : None
* Out parameters : None
* Return value   : function never returns
*********************************************************************/

int main (void)
{
  l_u8 cfg[20];
  l_u8 len = 0;
  l_bool configuration_ok = 0;
  l_bool stored_configuration = 0;

  /* Initialize the LIN interface */
  if (l_sys_init ()) {
    /* The init of the LIN software failed - call error routine */
  }
  /* Initialize the interface */
  if (l_ifc_init_i1 ()) {
    /* Initialization of the LIN interface failed - call error routine */
  }
  /* Now is the first time the LIN interrupts can be enabled */
  l_sys_irq_restore (INT_ENABLE_LEVEL);
  /* Configure the communication */
  configuration_ok = 0;
  stored_configuration = is_configuration_stored ();
  if (stored_configuration) {
    /* there is a stored configuration in NVRAM */
    read_from_NVRAM (cfg, &len);
    /* configure the communication */
    ld_set_configuration (i1, cfg, len);
    configuration_ok = 1;
  } else {
    /* wait for the master to configure me for 5 s*/
    l_u16 configuration_timeout = 1000;
    do {
      if (l_ifc_read_status_i1 () & SAVE_CONFIGURATION) {
        /* The master node is finished with the configuration */
        configuration_ok = 1;
        /* save configuration in NVRAM */
        ld_read_configuration (i1, cfg, len);
        write_to_NVRAM (cfg, len);
      }
      delay_5ms ();
      configuration_timeout--;
    } while (configuration_timeout || !configuration_ok);
  }
  if (!configuration_ok) {
    /* Timeout - no configuration from master, enter limp home */
  }

  while (1) {
    /* Call the only task */
    main_task ();
  }
  /* return code */
  return 1;
} /* main */
```

Bibliography

[1] ISO 14229-1, *Road vehicles — Unified diagnostic services (UDS) — Part 1: Specification and requirements*

[2] ISO 14229-2, *Road vehicles — Unified diagnostic services (UDS) — Part 2: Session layer services*

[3] ISO 14229-7, *Road vehicles — Unified diagnostic services (UDS) — Part 7: UDS on LIN implementation (UDSonLIN)*

[4] ISO 15765-2, *Road vehicles — Diagnostic communication over Controller Area Network (DoCAN) — Part 2: Transport protocol and network layer services*

[5] ISO 17987-2:2016, *Road vehicles — Local Interconnect Network (LIN) — Part 2: Transport protocol and network layer services*

[6] ISO 17987-3:2016, *Road vehicles — Local Interconnect Network (LIN) — Part 2: Protocol specification*

[7] ISO/IEC 7498-1, *Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model — Part 1*

[8] ISO/IEC 10731, *Information technology — Open Systems Interconnection — Basic Reference Model — Conventions for the definition of OSI services*

This page deliberately left blank

British Standards Institution (BSI)

BSI is the national body responsible for preparing British Standards and other standards-related publications, information and services.

BSI is incorporated by Royal Charter. British Standards and other standardization products are published by BSI Standards Limited.

About us

We bring together business, industry, government, consumers, innovators and others to shape their combined experience and expertise into standards -based solutions.

The knowledge embodied in our standards has been carefully assembled in a dependable format and refined through our open consultation process. Organizations of all sizes and across all sectors choose standards to help them achieve their goals.

Information on standards

We can provide you with the knowledge that your organization needs to succeed. Find out more about British Standards by visiting our website at bsigroup.com/standards or contacting our Customer Services team or Knowledge Centre.

Buying standards

You can buy and download PDF versions of BSI publications, including British and adopted European and international standards, through our website at bsigroup.com/shop, where hard copies can also be purchased.

If you need international and foreign standards from other Standards Development Organizations, hard copies can be ordered from our Customer Services team.

Copyright in BSI publications

All the content in BSI publications, including British Standards, is the property of and copyrighted by BSI or some person or entity that owns copyright in the information used (such as the international standardization bodies) and has formally licensed such information to BSI for commercial publication and use.

Save for the provisions below, you may not transfer, share or disseminate any portion of the standard to any other person. You may not adapt, distribute, commercially exploit, or publicly display the standard or any portion thereof in any manner whatsoever without BSI's prior written consent.

Storing and using standards

Standards purchased in soft copy format:

- A British Standard purchased in soft copy format is licensed to a sole named user for personal or internal company use only.
- The standard may be stored on more than 1 device provided that it is accessible by the sole named user only and that only 1 copy is accessed at any one time.
- A single paper copy may be printed for personal or internal company use only.

Standards purchased in hard copy format:

- A British Standard purchased in hard copy format is for personal or internal company use only.
- It may not be further reproduced – in any format – to create an additional copy. This includes scanning of the document.

If you need more than 1 copy of the document, or if you wish to share the document on an internal network, you can save money by choosing a subscription product (see 'Subscriptions').

Reproducing extracts

For permission to reproduce content from BSI publications contact the BSI Copyright & Licensing team.

Subscriptions

Our range of subscription services are designed to make using standards easier for you. For further information on our subscription products go to bsigroup.com/subscriptions.

With **British Standards Online (BSOL)** you'll have instant access to over 55,000 British and adopted European and international standards from your desktop. It's available 24/7 and is refreshed daily so you'll always be up to date.

You can keep in touch with standards developments and receive substantial discounts on the purchase price of standards, both in single copy and subscription format, by becoming a **BSI Subscribing Member**.

PLUS is an updating service exclusive to BSI Subscribing Members. You will automatically receive the latest hard copy of your standards when they're revised or replaced.

To find out more about becoming a BSI Subscribing Member and the benefits of membership, please visit bsigroup.com/shop.

With a **Multi-User Network Licence (MUNL)** you are able to host standards publications on your intranet. Licences can cover as few or as many users as you wish. With updates supplied as soon as they're available, you can be sure your documentation is current. For further information, email subscriptions@bsigroup.com.

Revisions

Our British Standards and other publications are updated by amendment or revision.

We continually improve the quality of our products and services to benefit your business. If you find an inaccuracy or ambiguity within a British Standard or other BSI publication please inform the Knowledge Centre.

Useful Contacts

Customer Services
Tel: +44 345 086 9001
Email (orders): orders@bsigroup.com
Email (enquiries): cservices@bsigroup.com

Subscriptions
Tel: +44 345 086 9001
Email: subscriptions@bsigroup.com

Knowledge Centre
Tel: +44 20 8996 7004
Email: knowledgecentre@bsigroup.com

Copyright & Licensing
Tel: +44 20 8996 7070
Email: copyright@bsigroup.com

BSI Group Headquarters

389 Chiswick High Road London W4 4AL UK