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)

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

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