LIN

Specification Package

Revision 1.2

The following Companies have provided advice for the contents of the Specification Package:
Audi AG, BMW AG, DaimlerChrysler AG, Motorola, Inc,
Volcano Communications Technologies AB, Volkswagen AG, Volvo Car Corporation.

Copies of this document may only be distributed by Audi, BMW, DaimlerChrysler, Motorola, Volcano Communications Technologies, Volkswagen, and Volvo Cars.

All distributions are registered.

The LIN Standard

LIN completes the portfolio of automotive communication protocol standards in the area of low cost networking.

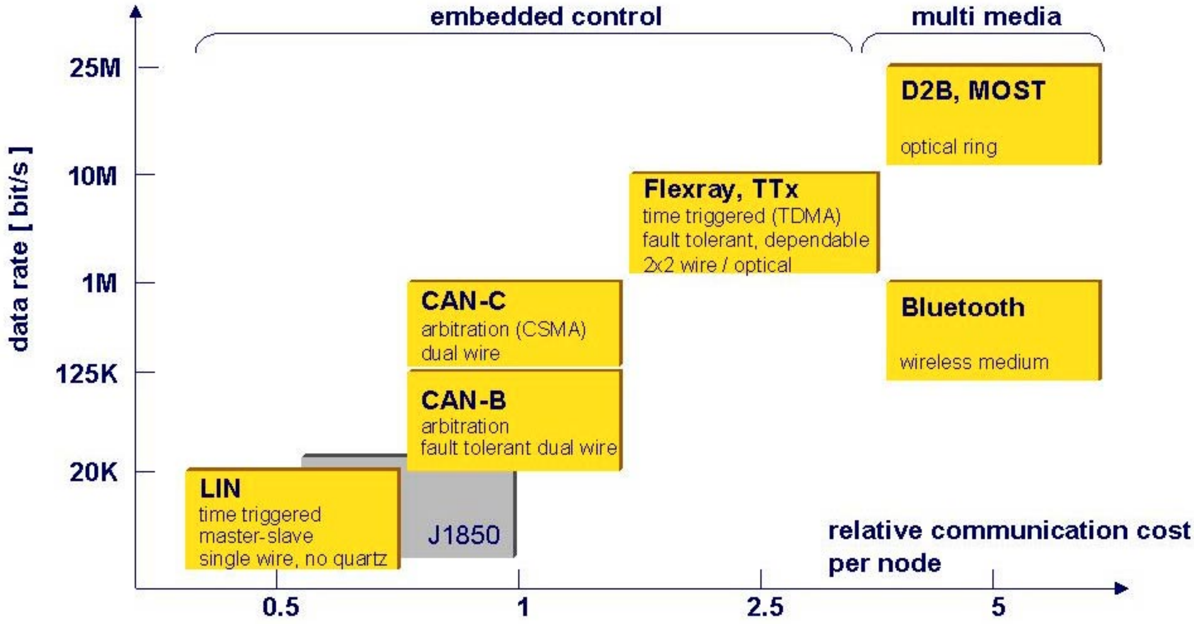


Figure 1: Major Network Protocols in Vehicles

The LIN Concept

LIN (Local Interconnect Network) is a concept for low cost automotive networks, which complements the existing portfolio of automotive multiplex networks. LIN will be the enabling factor for the implementation of a hierarchical vehicle network in order to gain further quality enhancement and cost reduction of vehicles. The standardization will reduce the manifold of existing low-end multiplex solutions and will cut the cost of development, production, service, and logistics in vehicle electronics.

The LIN standard includes the specification of the transmission protocol, the transmission medium, the interface between development tools, and the interfaces for software programming. LIN guarantees the interoperability of network nodes from the viewpoint of hardware and software, and a predictable EMC behavior.

This Specification Package consists of three main parts:

The **LIN Protocol Specification** describes the Physical Layer and the Data Link Layer of LIN.

The **LIN Configuration Language Description** describes the format of the LIN configuration file, which is used to configure the complete network and serve as a common interface between the OEM and the suppliers of the different network nodes, as well as an input to development and analysis tools.

The **LIN API** describes the interface between the network and the application program.

This concept allows the implementation of a seamless chain of development and design tools and enhances the speed of development and the reliability of the network.

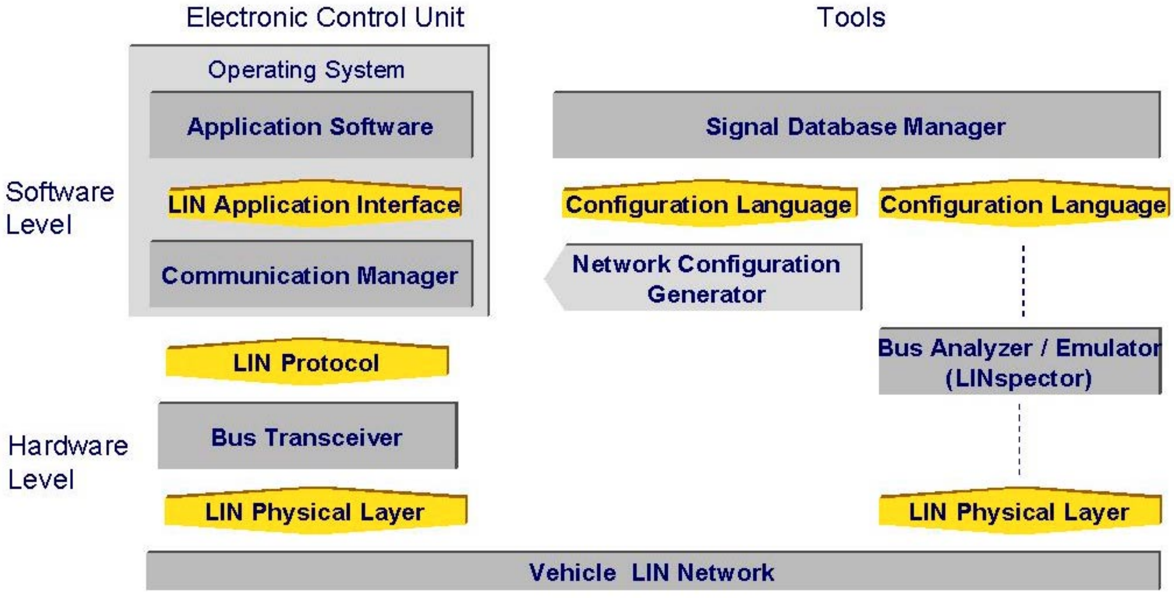


Figure 2: Scope of the LIN Specifications

LIN
Protocol Specification

Revision 1.2

The following companies have provided advice for the contents of the Protocol Specification:
Audi AG, BMW AG, DaimlerChrysler AG, Motorola, Inc,
Volcano Communications Technologies AB, Volkswagen AG, Volvo Car Corporation.

Copies of this document may only be distributed by Audi AG, BMW AG, DaimlerChrysler AG, Motorola, Inc, Volcano Communications Technologies AB, Volkswagen AG, Volvo Cars.

All distributions are registered.

Contact: H.-Chr. v. d. Wense, Motorola GmbH, Schatzbogen 7, D-81829 Munich, Germany
Ph: +49 (89) 92103-882 E-Mail: H.Wense@Motorola.com

1 INTRODUCTION

The LIN (Local Interconnect Network) is a serial communications protocol which efficiently supports the control of mechatronic nodes in distributed automotive applications. The domain is class-A multiplex buses with a single master node and a set of slave nodes [1].

The main properties of the LIN bus are:

- single-master / multiple-slave concept
- low cost silicon implementation based on common UART/SCI interface hardware, an equivalent in software, or as pure state machine.
- self synchronization without quartz or ceramics resonator in the slave nodes
- guarantee of latency times for signal transmission
- low cost single-wire implementation
- speed up to 20kbit/s.

The intention of this specification is to achieve compatibility with any two LIN implementations with respect to both the Data Link Layer and the Physical Layer according to the ISO/OSI Reference Model (see **Figure 2.1**).

LIN provides a cost efficient bus communication where the bandwidth and versatility of CAN are not required. The specification of the line driver/receiver follows the ISO 9141 standard [2] with some enhancements regarding the EMI behaviour.

1.1 REVISION HISTORY

July 5, 1999: Revision 1.0

April 17, 2000: Revision 1.1

November 17, 2000: Revision 1.2

- Protocol
  - **Table 2.1**: Corrected the unit of the transmission rate
  - Section 2: Connections: Termination resistance changed from range to typical values.
  - **Table 3.1**: Column with nominal values inserted
  - Section 3.1.3: Clarification about the use of response fields and the function of the checksum byte.
  - Section 3.2: Reserved additional identifier for general command and service messages and for future extended LIN revisions (upward compatibility); the command messages replace the former sleep mode message.
  - Section 3.3: Clarification of calculation of frame lengths
  - **Table 3.4**: Column for nominal values added
  - Section 6.1: Error handling of identifier parity error removed
  - Section 6.1: Checksum Error corrected
  - **Table 8.1**: Clock tolerances for master and slave nodes with resonator now specified.
- Physical Layer
  - **Table 10.3**: Maximum slew rate specified.
  - **Table 10.4**: Changed C_{SLAVE} and C_{MASTER} for better ESD and EMI behaviour ().
  - Section 10.4: Note on ESD level modified.
- All Document: Replaced “Data Frame” with “MESSAGE FRAME” or appropriate wording.

1.2 CONTRIBUTORS

To this specification have contributed:

J. Bauer, V. Riebeling, Audi AG, Ingolstadt.

J. Fröschl, M. Kaindl, Dr. J. Krammer, BMW AG, Munich.

C. Bracklo, W. Welja, DaimlerChrysler AG, Stuttgart.

R. Erckert, Dr. J. Krücken, Dr. A. Krüger, Dr. W. Specks, H.-C. Wense,
Motorola GmbH, Munich.

I. Horváth, A. Rajnák, Volcano Communications Technologies, Gothenburg.

J. Ende, T. Zawade, Volkswagen AG

L. Casparsson, Volvo Car Corporation, Gothenburg.

The implementation of any aspect of this specification may be protected by intellectual property rights.

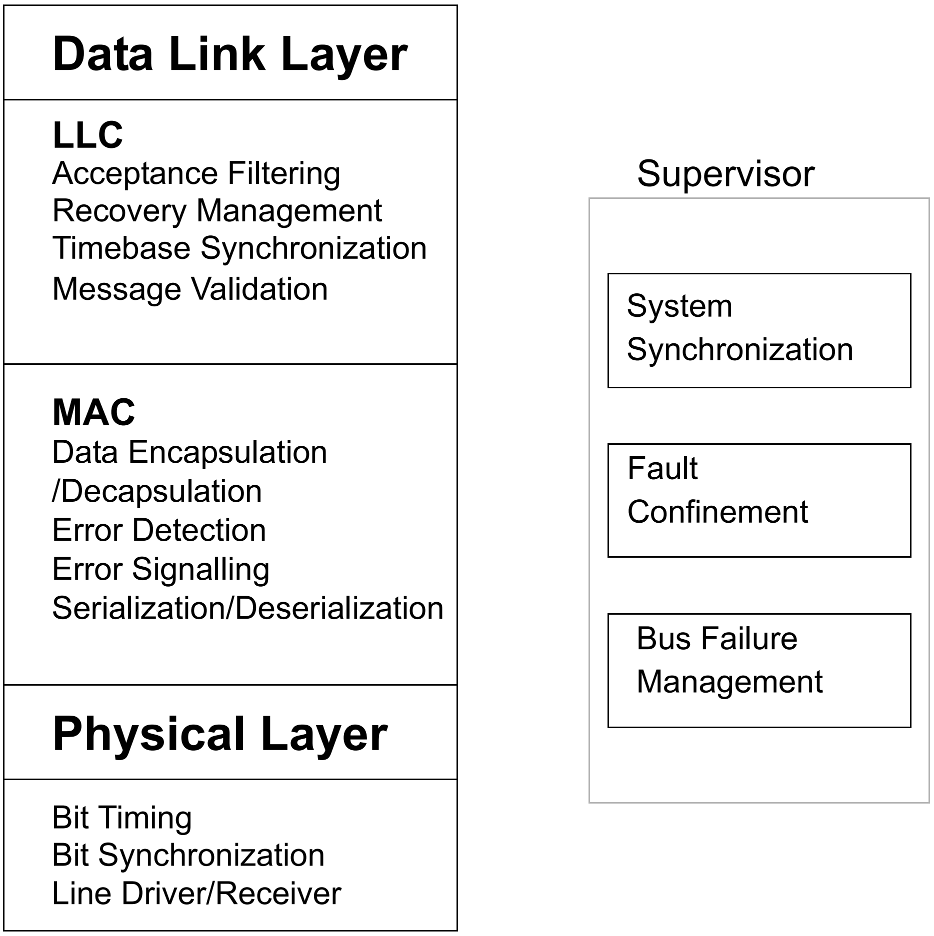
2 BASIC CONCEPTS

LIN protocol has the following properties

- single-master, multiple-slave organization (i.e. no bus arbitration)
- guarantee of latency times for signal transmission
- selectable length of MESSAGE FRAME: 2, 4, and 8 byte
- configuration flexibility
- multi-cast reception with time synchronization, without quartz or ceramics resonator in slave nodes
- data-checksum security, and error detection;
- detection of defect nodes in the network.
- minimum cost for semiconductor components (small die-size, single-chip systems)

The layered architecture of LIN according to the OSI Reference Model is shown in **Figure 2.1**.

- The Physical Layer defines how signals are actually transmitted over the bus medium. Within this specification the driver/receiver characteristics of the Physical Layer are defined.
- The MAC (Medium Access Control) sublayer represents the kernel of the LIN protocol. It presents messages received from the LLC sublayer and accepts messages to be transmitted to the LLC sublayer. The MAC sublayer is supervised by a management entity called Fault Confinement.
- The LLC (Logical Link Control) sublayer is concerned with Message Filtering and Recovery Management.



LLC = Logical Link Layer
MAC = Medium Access Control

Figure 2.1: OSI Reference Model

The scope of this specification is to define the Data link Layer and the Physical Layer and the consequences of the LIN protocol on the surrounding layers.

Messages

Information on the bus is sent in fixed format messages of selectable length (see Section 3). Every MESSAGE FRAME comprises between two, four, or eight bytes of data plus three bytes of control and safety information. The bus traffic is controlled by a single master. Each message frame starts with a break signal and is followed by a synchronization field and an identifier field, all sent out by a master task. The slave task sends back the data field and the check field (see **Figure 2.2**).

Data can be sent from the master control unit to any slave control unit by the slave task in the master control unit. A slave-to-slave communication can be triggered by a corresponding message ID from the master.

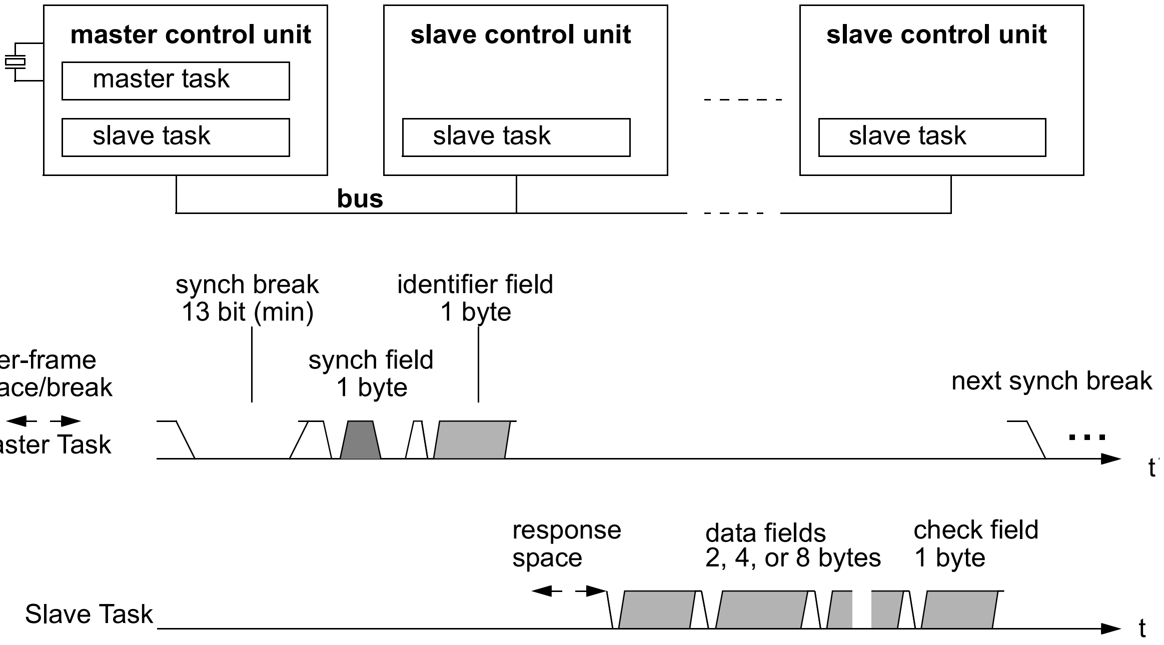


Figure 2.2: Communication Concept of LIN

Information Routing

In LIN systems a node does not make use of any information about the system configuration, except for the denomination of the single master node.

System Flexibility: Nodes can be added to the LIN network without requiring hardware or software changes in other slave nodes.

Message routing: The content of a message is named by an IDENTIFIER. The IDENTIFIER does not indicate the destination of the message, but describes the meaning of the data. The maximum number of identifier is 64, out of which 4 are reserved for special communication purposes such as software upgrades or diagnostics.

Multicast: As a consequence of the Message Filtering any number of nodes can simultaneously receive and act upon messages.

Bit rate

The maximum baud rate is 20kbit/s, given by the EMI limitation of the single wire transmission medium. The minimum baud rate is 1kbit/s to avoid conflicts with the practical implementation of time-out periods.

In order to allow implementation of low cost LIN devices, the use of following bit-rates is recommended:

Slow	**Medium**	**Fast**
2400 bit/sec	9600 bit/sec	19200 bit/sec

Table 2.1: Recommended Bit Rates

Single-Master - No Arbitration

Only the controller node containing the Master Task is allowed to transmit the message header, and one Slave Task responds to this header. As there is no arbitration procedure, an error occurs if more than one slave responds. The fault confinement for this case has to be specified by the user depending on the application requirements.

Safety

Error Detection

- monitoring, the transmitter compares ‘should’ with ‘is’ value on the bus
- inverted modulo-256 checksum for the Data fields, with the carry of the MSB being added to the LSB
- double-parity protection for the identifier field

Performance of Error Detection

- all local errors at the transmitter are detected
- high error coverage of global protocol errors.

Error Signalling and Recovery time

The direct signalling of errors is not possible due to the single-master concept. Errors are locally detected and provided in form of diagnostics on request (see Section 6).

Fault Confinement

LIN nodes are able to distinguish short disturbances from permanent failures and carry out appropriate local diagnostics and actions on failures (see Section 7).

Connections

The maximum number of nodes in a LIN network is not only limited by the number of identifiers (see Information Routing above) but also by the physical properties of the bus line:

- Recommendation: The number of nodes in a LIN network should not exceed 16. Otherwise the reduced network impedance may prohibit a fault free communication under worst case conditions. Every additional node lowers the network impedance by approximately 3% (30 kΩ || ~1 kΩ).
- The accumulated ‘galvanic’ wire length in a network is less or equal 40 m.
- The bus termination resistance is typically 1kΩ for the master node and 30kΩ for the slave nodes.

Single Channel

The bus consists of a single channel that carries bits. From this data resynchronization information can be derived.

Physical Layer

The Physical Layer is a single line, wired-AND bus with pull-up resistors in every node, being supplied from the vehicle power net (VBAT), see Section 10. A diode in series with the pull-up resistor prevents the electronic control unit (ECU) from being powered by the bus in case of a local loss of battery.

The signal shaping is defined by the requirements of EMI and clock synchronization.

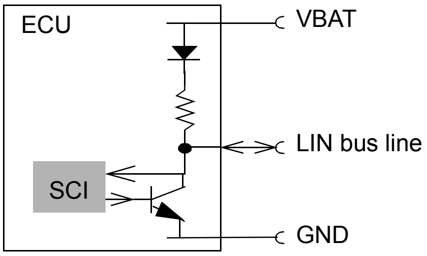


Figure 2.3: Illustration of the Physical Layer

Bus Values

The bus can have two complementary logical values: ‘dominant’ or ‘recessive’. The correspondence with bit and voltage values are given in **Table 2.2**.

logical value	bit value	bus voltage (Section 10.2)
dominant	0	ground
recessive	1	battery

Table 2.2: Logical and Physical Bus Values

Acknowledgment

An acknowledgment procedure for a correctly received message is not defined in the LIN protocol. The master control unit checks the consistency of a message being initiated by the master task and being received by it’s own slave task. In case of inconsistency (e.g. missing slave response, incorrect checksum etc.) the master task can change the message schedule.

In case a slave has detected an inconsistency, the slave controller will save this information and provide it on request to the master control unit in form of diagnostics information. This diagnostics information can be transmitted as data in a regular MESSAGE FRAME.

Command Frame and Extended Frame

Four identifier from the set with 8-byte response are reserved for particular message frames: two command frames and two extended frames.

The two command frames include an 8-byte response and are used for data up- and downloads from the master to slave nodes and vice versa. This feature is used for software updates, network configuration, and diagnostic purposes. The frame structure is identical with a regular message. The response fields contain user-defined command fields instead of data fields that put the slave nodes for example in a service mode or in the sleep mode.

Two extended frame identifiers are reserved to allow the embedding of user-defined message formats and future LIN formats into the current LIN protocol without violating the current LIN specification. This ensures the upward compatibility of LIN slaves to future revisions of the LIN protocol. The extended frame identifier announce an unspecified frame format to all bus participants. The identifier can be followed by an arbitrary number of LIN byte fields. A slave receiving such an identifier must ignore all subsequent byte fields until the next synchronization break.

Sleep Mode / Wake-Up

To reduce the system's power consumption, a LIN node may be sent into the sleep mode without any internal activity and with passive bus driver. The message that is used to broadcast a sleep mode is a dedicated command as defined in Section 3.2. The bus is recessive during sleep mode.

The sleep mode is finished with a wake-up by any bus activity or by internal conditions in any bus node. In case of a node-internal wake-up, a procedure based on use of the WAKE-UP SIGNAL shall be used for alerting the master. A Wake-Up Frame is a monotonous sequence of dominant bits as specified in Section 3.4.

On wake-up, the internal activity is restarted, although the MAC sublayer will be waiting for the systems oscillator to stabilize and - in the case of a slave node - it will then wait until it has synchronized itself to the bus activity (by waiting for the dominant synchronization break) before the participation in the bus communication is resumed.

Clock Recovery and SCI Synchronization

Each Message Frame starts with a Synchronization BREAK (SYNCH BREAK) followed by a synchronization field (SYNCH FIELD) which includes five falling edges (i.e. transition 'recessive' to 'dominant') in multiple distances of the bit-time. This distance can be measured (i.e. by a timer capture function) and be used to calculate the node-internal time base of the slaves (see Section 3.1 and Section 9).

The synchronization break frame enables those slave nodes which have lost synchronization to identify the synchronization field (see Section 3.1.2).

Oscillator Tolerance

The Bit Timing requirements allow the usage of pre-trimmed on-chip oscillators in the slave nodes with the tolerances given in **Table 8.1**. The clock base in the master node is given by a quartz or ceramic resonator, and does represent the 'frequency center point'.

3 MESSAGE TRANSFER

3.1 MESSAGE FRAME

Message transfer is manifested and controlled by one MESSAGE FRAME format. A MESSAGE FRAME carries synchronization and identifier information from the master task to the slave tasks and a data information from one slave task to all other slave tasks. The master task resides in the master node and is responsible for the schedule of the messages: It sends the HEADER of the MESSAGE FRAME. The slave tasks reside in all (i.e. master and the slave) nodes, one of them (either the master node or one slave node) sending the RESPONSE of the message.

A MESSAGE FRAME (**Figure 3.1**) is composed of a HEADER which is sent by the master node and a RESPONSE that is sent by either the master node or one of the slave nodes. The HEADER consists of a SYNCH BREAK FIELD, SYNCHRONIZATION FIELD (SYNCH FIELD) and IDENTIFIER FIELD. The RESPONSE consists of three to nine BYTE FIELDS: two, four, or eight DATA FIELD's, and one CHECKSUM FIELD. The BYTE FIELD's are separated by interbyte spaces, HEADER and RESPONSE are separated by one in-frame-response space. The minimum length of interbyte spaces and the in-frame-response space is zero. The total maximum length of these spaces is limited by the maximum length of the Message Frame T_{FRAME_MAX} as specified in **Table 3.3**.

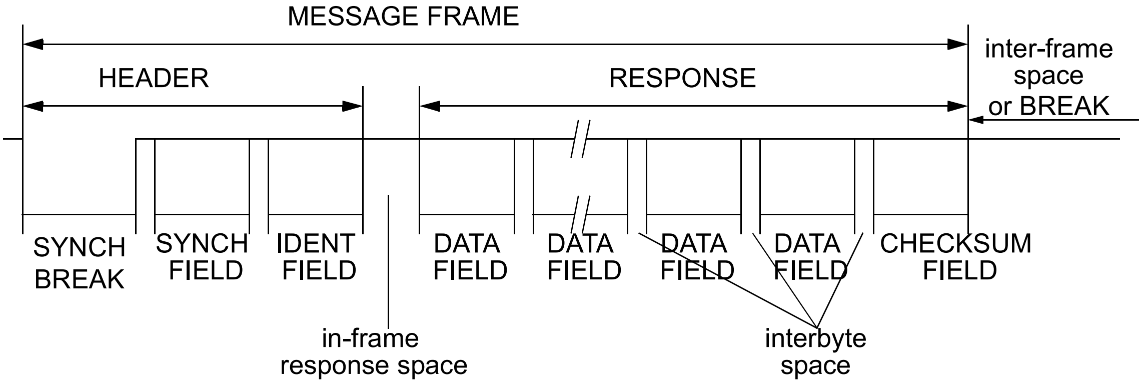


Figure 3.1: LIN MESSAGE FRAME

3.1.1 BYTE fields

The BYTE FIELD format (**Figure 3.2**) is commonly known as ‘SCI’ or ‘UART’ serial data format (8N1-coding). Every BYTE FIELD has a length of ten BIT TIMES. The START BIT marks the begin of the BYTE FIELD and is ‘dominant’. It is followed by eight DATA BIT’s with the LSB first. The STOP BIT marks the end of the BYTE FIELD and is ‘recessive’.

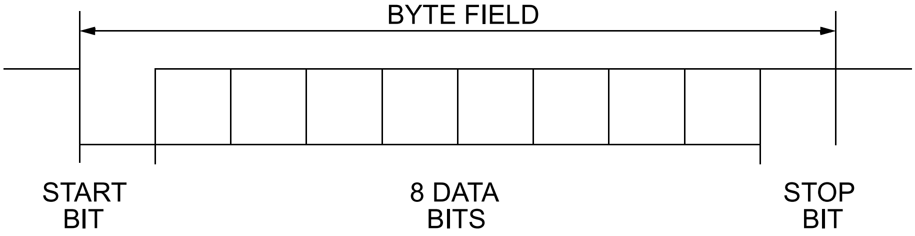


Figure 3.2: LIN BYTE FIELD

3.1.2 HEADER fields

SYNCHRONISATION BREAK

In order to identify clearly a beginning of a message frame it’s first field is a synchronization break (SYNCH BREAK). A SYNCH BREAK FIELD is always sent by the master task.

This provides a regular opportunity for slave tasks to synchronize on the bus clock.

The synchronization break field consists of two different parts (see **Figure 3.3**). The first part consists of a dominant bus value with the duration of T_{SYNBRK} or longer (i.e. (minimum T_{SYNBRK}, not necessarily exactly). The following second part is the recessive synchronization delimiter with a minimum duration of T_{SYNDEL}. This second field is necessary to enable the detection of the start bit of the following SYNCH FIELD.

The maximum length of break and delimiter is not explicitly specified but must fit into the gross time budget given for the entire message header T_{HEADER_MAX}, as specified in **Table 3.3**)

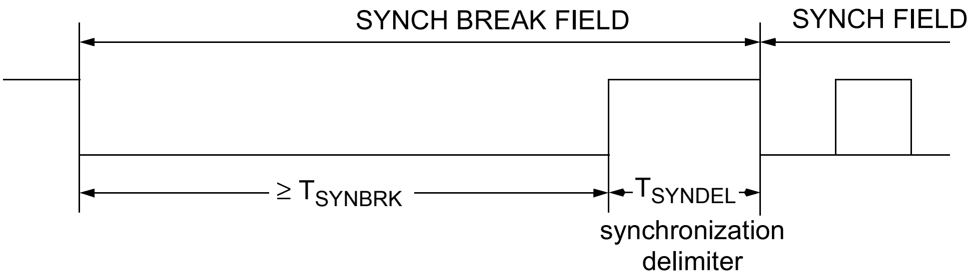


Figure 3.3: SYNCH BREAK FIELD

The timing specifications for the SYNCH BREAK FIELD and for its evaluation by slave control units are a result of the clock tolerances that are allowed in a LIN network (see **Table 8.1**). A dominant signal is recognized as SYNCH BREAK FIELD if it is longer than the maximum regular sequence of dominant bits in the protocol (here: a '0x00' field with 9 dominant bits). A slave node detects such a break if its duration surpasses the period of T_{SBRKTS}, measured in slave bit times (see **Table 3.1**). This 'threshold' results from the maximum local clock frequency being specified for slave nodes. There are specified two values for the threshold T_{SBRKTS}, depending on the accuracy of the local time base.

The dominant length of the SYNCH BREAK FIELD has to be at minimum T_{SYNBRK} (but can be longer), measured in master bit times. This minimum value results from the required 'threshold' value in conjunction with the minimum local clock frequency being specified for slave nodes (see **Table 8.1**).

SYNCH BREAK FIELD	**LOGICAL**	**NAME**	**MIN [T_{bit}]**	**Nom [T_{bit}]**	**MAX [T_{bit}]**
SYNCH BREAK LOW PHASE	dominant	T_{SYNBRK}	13[a]		-
SYNCH BREAK DELIMITER	recessive	T_{SYNDEL}	1[a]		-
SYNCH BREAK THRESHOLD SLAVE	dominant	T_{SBRKTS}		11[b]	
				9[c]	

Table 3.1: Timing of the SYNCH BREAK FIELD.

[a] This bit time is based on the master time base.

[b] This bit time is based on the local slave time base. It is valid for nodes with a clock tolerance lower than $F_{TOL\_UNSYNCH}$ (see **Table 8.1**), e.g. for slave nodes with RC oscillator.

[c] As in b. but valid for nodes with a clock tolerance lower than F_{TOL_SYNCH}, e.g. for slave nodes with quartz or ceramic resonator (see **Table 8.1**).

SYNCH FIELD

The SYNCH FIELD contains the information for the clock synchronization. The SYNCH FIELD is the pattern ’0x55’ which is characterized by five falling edges (i.e. ‘recessive’ to ‘dominant’ edges) within eight bit times distance (**Figure 3.4**). The synchronization procedure is defined in Section 9.

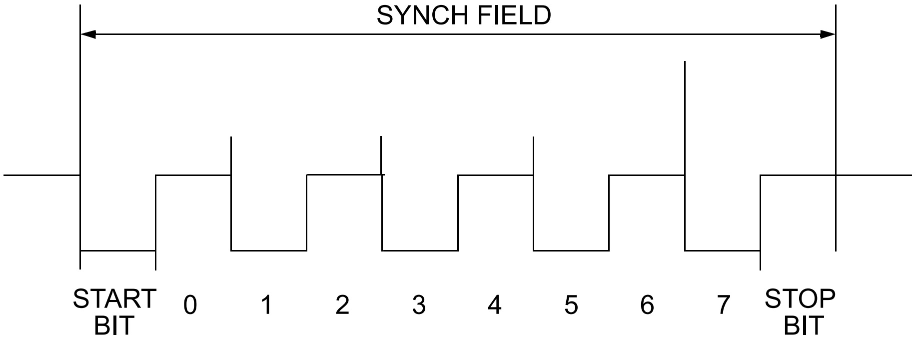


Figure 3.4: SYNCH FIELD

IDENTIFIER FIELD

The IDENTIFIER FIELD (ID-Field) denotes the content and length of a message. The content is represented by six IDENTIFIER bits and two ID PARITY bits (**Figure 3.5**). The IDENTIFIER bits ID4 and ID5 define the number of data fields N_{DATA} in a message (**Table 3.2**). This divides the set of 64 identifiers in four subsets of sixteen identifiers, with 2, 4, and 8 data fields, respectively.

ID5	**ID4**	**N_{DATA} (number of data fields) [byte]**
0	0	2
0	1	2
1	0	4
1	1	8

Table 3.2: Control of the Number of Data Fields in a MESSAGE FRAME

Identifier with identical ID-bits ID0..ID3 but different length codes ID4, ID5 denote distinguished messages.

Note: A message length coding different from the specification in Table 3.2 might be chosen, if stringent technical issues require this (e.g. in climate systems). In such a case, the number of data bytes can be arbitrarily chosen from zero through eight, independently from the identifier.

The parity check bits of the identifier are calculated by a mixed-parity algorithm:

$$P0 = ID0 \oplus ID1 \otimes ID2 \oplus ID4 \text{ (even parity)}$$

$$P1 = \overline{ID1 \oplus ID3 \otimes ID4 \oplus ID5} \text{ (odd parity)}$$

This way no pattern with all bits recessive or dominant will be possible.

The identifiers 0x3C, 0x3D, 0x3E, and 0x3F with their respective IDENTIFIER FIELDS 0x3C, 0x7D, 0xFE, and 0xBF (all 8-byte messages) are reserved for command frames (e.g. sleep mode) and extended frames (Section 3.2).

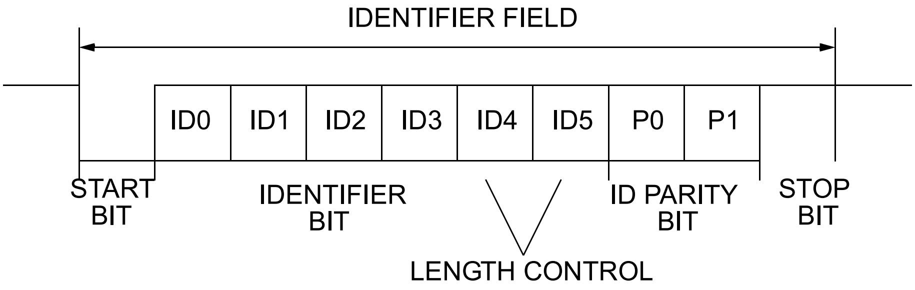


Figure 3.5: IDENTIFIER FIELD

3.1.3 RESPONSE fields

Depending on the application, the response fields of a message (data and checksum) can but need not to be processed if this information is irrelevant for the control unit. This is for example the case for unknown or corrupted identifier. In such a case, the checksum calculation can be omitted (see also Appendix A.5).

DATA FIELD

The DATA FIELD consists of a BYTE FIELD containing eight bits of data to be transferred by a MESSAGE FRAME. The transmission happens LSB first (**Figure 3.6**).

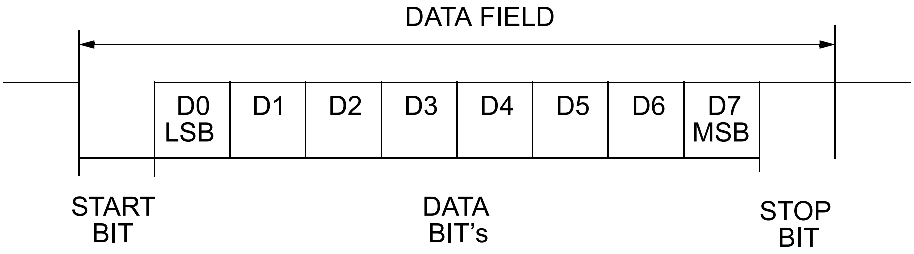


Figure 3.6: DATA FIELD

<u>CHECKSUM FIELD</u>

The CHECKSUM FIELD contains the inverted modulo-256 sum over all data bytes (**Figure 3.7**). The sum is calculated by “ADD with Carry” where the carry bit of each addition is added to the LSB of it’s resulting sum. This guarantees security also for the MSBs of the data bytes.

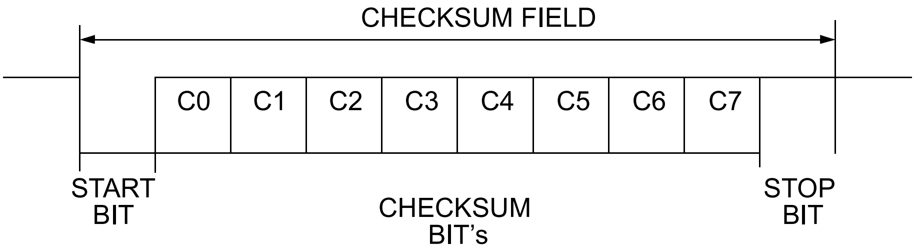


Figure 3.7: CHECKSUM FIELD

The sum of modulo-256 sum over all data bytes and the checksum byte must be ’0xFF’.

3.2 RESERVED IDENTIFIERS

<u>COMMAND FRAME IDENTIFIER</u>

Two COMMAND FRAME IDENTIFIER are reserved to broadcast general command requests for service purposes from the master to all bus participants. The frame structure is identical to that of a regular 8-byte MESSAGE FRAME (see **Figure 3.8**) and is distinguished only by the reserved identifiers

’0x3C’ ID-Field = 0x3C; ID0,1,6,7 = 0; ID 2,3,4,5 = 1 Master request frame, and

’0x3D’ ID-Field = 0x7D; ID1,7 = 0; ID 0,2,3,4,5,6 = 1 Slave Response frame, (see also **Appendix A2**).

The identifier ’0x3C’ is a ’Master Request-frame’ (MasterReq) to send commands and data from the master to the slave node. The identifier ’0x3D’ is an ’Slave Response-frame’ (SlaveResp) that triggers one slave node (being addressed by a prior download-frame) to send data to the master node.

The Command Frames with their first DATA FIELD containing a value from 0x00 to 0x7F are reserved, their usage will be defined by the LIN consortium. The remaining command frames can be assigned by the user.

First data byte of command frame: bit D7 = 0: reserved usage
bit D7 = 1: free usage

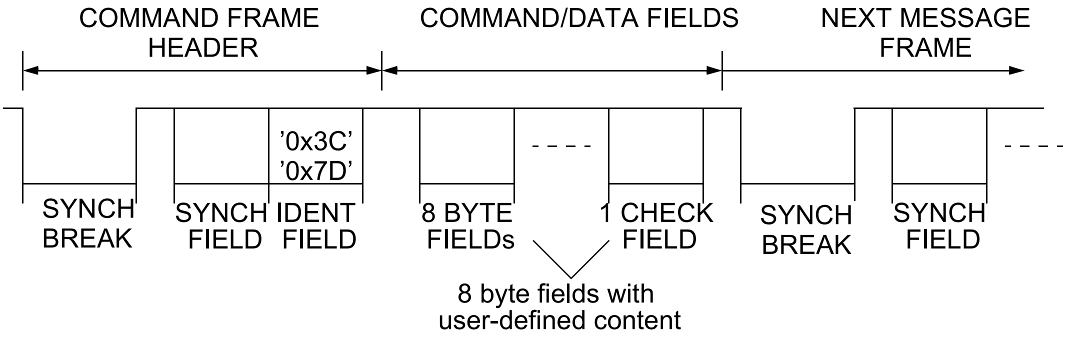


Figure 3.8: LIN COMMAND FRAME

SLEEP MODE COMMAND

The SLEEP MODE COMMAND is used to broadcast the sleep mode to all bus nodes. There is no more bus activity after completion of this message until a WAKE-UP SIGNAL on the bus ends the sleep mode (Section 3.4). The SLEEP MODE COMMAND is a download COMMAND FRAME with the first DATA FIELD being 0x00.

EXTENDED FRAME IDENTIFIER

Two extended frame identifiers are reserved to allow the embedding of user-defined message formats and future LIN formats into the LIN protocol without violating the current LIN specification. This ensures the upward compatibility of LIN slaves with future revisions of the LIN protocol.

The extended frames are distinguished by the reserved IDENTIFIER FIELDs

’0x3E’ ID-Field = 0xFE; ID0 = 0; ID 1,2,3,4,5,6,7 = 1 user-defined extended frame, and

’0x3F’ ID-Field = 0xBF; ID6 = 0; ID 0,1,2,3,4,5,7 = 1 future LIN extension (see **Appendix A2**).

The identifier ’0x3E’ (IDENTIFIER FIELD = ’0xFE’) indicates a user defined extended frame which is free for use. The identifier ’0x3F’ (IDENTIFIER FIELD = ’0xBF’) is strictly reserved for the occasion of a future extended version of LIN[a] and must not be used in current implementations.

The identifier can be followed by an arbitrary number of LIN BYTE FILEDS (see **Figure 3.9**). The frame length, the communication concept[b], and the data content are not specified here. The length coding of the ID field does not apply to these two frames.

A slave receiving the EXTENDED FRAME IDENTIFIER and not being in the position to make use of the content, must ignore all subsequent LIN BYTE FIELDs until the reception of the next SYNCH BREAK.

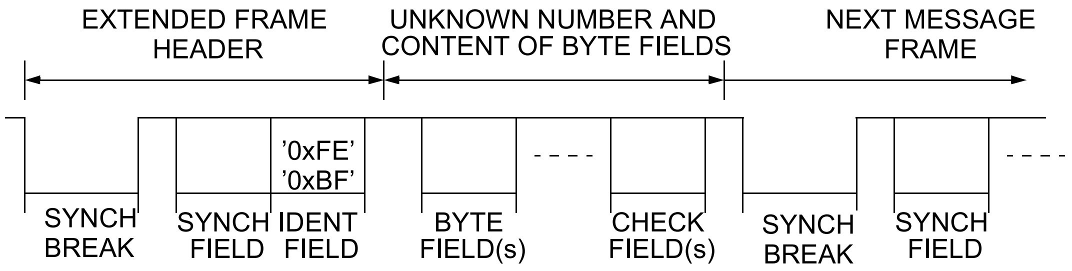


Figure 3.9: LIN EXTENDED FRAME

3.3 LENGTH OF MESSAGE FRAME AND BUS SLEEP DETECT

MESSAGE FRAMES start with a SYNCH BREAK FIELD and end with the CHECKSUM FIELD. The BYTE-FIELDS within a MESSAGE FRAME are separated by interbyte-spaces and the in-frame response space. The lengths for interbyte-spaces and in-frame response space are not defined, only the total length of a Message Frame is limited. The minimum Frame length T_{FRAME_MIN} is the minimum time needed to transmit a complete frame (lenghts of interbyte-spaces and in-frame response space = 0). The maximum Frame length T_{FRAME_MAX} is the maximum time allowed for the trans-

Note a: comparable to the switch from ’standard’ to ’extended’ format in the CAN protocol [3].
Note b: could be even multi-master

mission of the frame. The values are given in **Table 3.3**. They are dependent on the number of Data Byte Fields N_{DATA} and do not include system inherent (i.e. physical) signal delays.

TIME	NAME	Time [T_{bit}]
Minimum Length of Message Frame	T_{FRAME_MIN}	$10 * N_{DATA} + 44$
Minimum Length of Header	T_{HEADER_MIN}	34
Maximum Length of Header	T_{HEADER_MAX}	$(T_{HEADER\_MIN} + 1^{a}) * 1.4$
Maximum Length of Message Frame	T_{FRAME_MAX}	$(T_{FRAME\_MIN} + 1^{a}) * 1.4$
Bus Idle Time-Out	T_{TIME_OUT}	25,000

Table 3.3: Timing of a Message Frame

[a] The term '+1' leads to an integer result for T_{HEADER_MIN} and T_{FRAME_MAX}

If a slave detects no activity on the bus for T_{TIME_OUT}, it may assume that the bus is in sleep mode. This could e.g. be the case when the sleep message was corrupted.

3.4 WAKE-UP SIGNAL

The sleep mode of the bus can be terminated by any node by sending a WAKE-UP SIGNAL. A wake-up signal can be sent by any slave task but only if the bus was previously in sleep mode and a node-internal request for wake-up is pending.

The wake-up signal consists of the character '0x80'. When the slave is not in synch with the master node the signal can be 15% longer or shorter than a signal based on an accurate clock source. The character '0x80' will be detected by the master as a valid data byte, either as '0xC0', '0x80', or '0x00'. The first field is given by a sequence of T_{WUSIG} dominant bits, i.e. 8 dominant bits (including start bit). The following second field is the recessive wake-up delimiter with a duration of at least T_{WUDEL}, i.e. at least 4 Bit times (including stop-bit and a recessive pause).

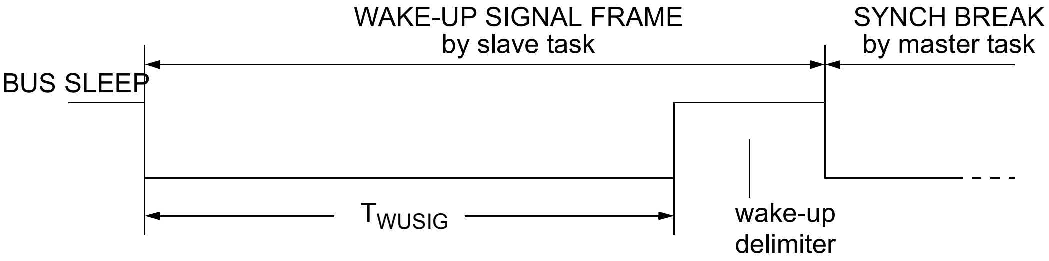


Figure 3.10: WAKE-UP SIGNAL FRAME

After a wake-up signal has been sent to the bus, all nodes run through the Start-Up procedure and wait for the master task to send a SYNCH BREAK FIELD followed by the SYNCH FIELD. If no SYNCH FIELD is detected before TIME-OUT AFTER WAKE-UP SIGNAL, a new WAKE-UP SIGNAL is issued by the node requesting the first WAKE-UP. This sequence is issued not more than three times. Then the transmission of WAKE-UP SIGNALS is suspended for a TIME-OUT AFTER THREE BREAKS as specified in **Table 3.4** and illustrated by Appendix A.1. The re-transmission of a WAKE-UP SIGNAL is allowed only to the node which has an internal request for wake-up pending. After three re-transmissions of the WAKE-UP SIGNAL after a TIME-OUT AFTER THREE BREAKS the application has to decide whether it stops retransmitting.

WAKE-UP	**LOGICAL**	**NAME**	**MIN [T_{bit}]**	**Nom [T_{bit}]**	**MAX [T_{bit}]**
WAKE-UP SIGNAL	dominant	T_{WUSIG}		8[a]	
WAKE-UP SIGNAL DELIMITER	recessive	T_{WUDEL}	4[b]		64
TIME-OUT AFTER WAKE-UP SIGNAL	recessive	T_{TOBRK}			128
TIME-OUT AFTER THREE BREAKS	recessive	T_{T3BRK}	15,000		

Table 3.4: WAKE-UP SIGNAL Timing.

[a] This bit time is based on the respective Slave Clock

[b] To be checked if this is sufficient as start-up time for any node in the network

If not otherwise noted, the bit-time $\mathbf{T_{bit}}$ refers to the SCI baud rate of the master node (see Section 9).

4 MESSAGE FILTERING

Message filtering is based upon the whole identifier. It has to be ensured by network configuration that not more than one slave task is responding on a transmitted identifier.

5 MESSAGE VALIDATION

The message is valid for both the transmitter and the receiver if there is no error detected until the end of frame.

If a message is corrupted, this message is regarded by the master and the slave tasks as not transmitted.

Note

The actions that master and slave tasks undertake upon transmission and reception of a corrupted message are not part of the protocol specification. Actions such as re-transmission by the master or fall-back operations by the slaves depend strongly on the application requirements and shall be specified in the application layer.

Event information that is sent over the bus may be lost without being detected.

6 ERROR AND EXCEPTION HANDLING

6.1 ERROR DETECTION

There are five different message error types specified. Causes of errors are listed in Appendix A.4:

Bit-Error

A unit that is sending a bit on the bus also monitors the bus. A BIT_ERROR has to be detected at that bit time, when the bit value that is monitored is different from the bit value that is sent.

Checksum-Error

A CHECKSUM_ERROR has to be detected if the sum of the inverted modulo-256 sum over all received data bytes and the checksum does not result in ’0xFF’ (see Section 3.1, CHECKSUM FIELD).

Identifier-Parity-Error

A parity error in the identifier (i.e. corrupted identifier) will not be flagged. Typical LIN slave applications do not distinguish between an unknown but valid identifier, and a corrupted identifier. However, it is mandatory for all slave nodes to evaluate in case of a known identifier all 8 bits of the ID-Field and distinguish between a known and a corrupted identifier.

Slave-Not-Responding-Error

A NO_RESPONSE_ERROR has to be detected if the MESSAGE FRAME is not fully completed within the maximum length T_{FRAME_MAX} (see Section 3.3) by any slave task upon transmission of the SYNCH and IDENTIFIER fields.

Inconsistent-Synch-Field-Error

An Inconsistent-Synch-Field-Error has to be detected if a slave detects the edges of the SYNCH FIELD outside the given tolerance (see Section 8).

No-Bus-Activity

A No-Bus-Activity condition has to be detected if no valid SYNCH BREAK FIELD or BYTE FIELD was received for more than $T_{TIMEOUT}$ (see Section 3.3) since the reception of the last valid message.

6.2 ERROR SIGNALLING

Detected errors are not signalled by the LIN protocol. Errors are flagged within each bus node and must be accessible to the fault confinement procedures that are specified in Section 7.

7 FAULT CONFINEMENT

The concept of fault confinement relies mainly on the master node that shall handle as much as possible of error detection, error recovery, and diagnostics. Fault confinement strongly depends on the system requirements and is thus not part of the LIN protocol except for some minimum features. See Appendix A.4 for possible error causes and Appendix A.5 for proposed confinement procedures.

MASTER CONTROL UNIT

The master control unit shall detect the following error situations:

- Master task sending: A Bit-Error or Identifier-Parity-Error in synchronization or identifier byte is detected while reading back the own transmission.
- Slave task in the master control unit receiving: A Slave-Not-Responding-Error or a Checksum-Error is detected when expecting or reading a data from the bus.

SLAVE CONTROL UNIT

Any slave control unit shall detect the following error situations:

- Slave task sending: A Bit-Error in a data or checksum field while reading back the own transmission.
- Slave task receiving: An Identifier-Parity-Error or a Checksum-Error is detected while reading from the bus.

  A Slave-Not-Responding-Error is detected while reading from the bus.
  This error type must be detected when a slave expects a message from another slave (depending on the identifier) but no valid message appears on the bus within the time frame given by the maximum length of the message frame T_{FRAME_MAX} as specified in Table 3.3. When a slave does not expect a message (depending on the identifier) it does not need to detect this error.

  An Inconsistent-Synch-Byte-Error is detected when the edges of the SYNCH FIELD are not detected within the given tolerance (see Section 8).

8 OSCILLATOR TOLERANCE

On-chip clock generators can achieve a frequency tolerance of better than ± 15% with internal-only calibration. This accuracy is sufficient to detect a synchronization break in the message stream. The subsequent fine calibration using the synchronization field ensures the proper reception and transmission of the message. The on-chip oscillator must be stable for the rest of the message, taking into account the impacts of temperature and voltage drift during the operation.

clock tolerance	**Name**	**$\Delta F / F_{Master}$**
master node	$F_{TOL\_RES\_MASTER}$	$< \pm 0.5\%$
slave node with quartz or ceramic resonator (without the need to synchronize)	$F_{TOL\_RES\_SLAVE}$	$< \pm 1.5\%$
slave without resonator, lost synchronization	$F_{TOL\_UNSYNCH}$	$< \pm 15\%$
slave without resonator, synchronized and for a complete message	F_{TOL_SYNCH}	$< \pm 2\%$

Table 8.1: Oscillator Tolerance

9 BIT TIMING REQUIREMENTS AND SYNCHRONIZATION PROCEDURE

9.1 BIT TIMING REQUIREMENTS

If not otherwise stated, all bit times in this document use the bit timing of the Master Node as a reference.

9.2 SYNCHRONIZATION PROCEDURE

The SYNCH FIELD consists of the pattern ’0x55’. The synchronization procedure has to be based on time measurement between falling edges of the pattern. The falling edges are available in distances of 2, 4, 6 and 8 bit times which allows a simple calculation of the basic bit times T_{bit}.

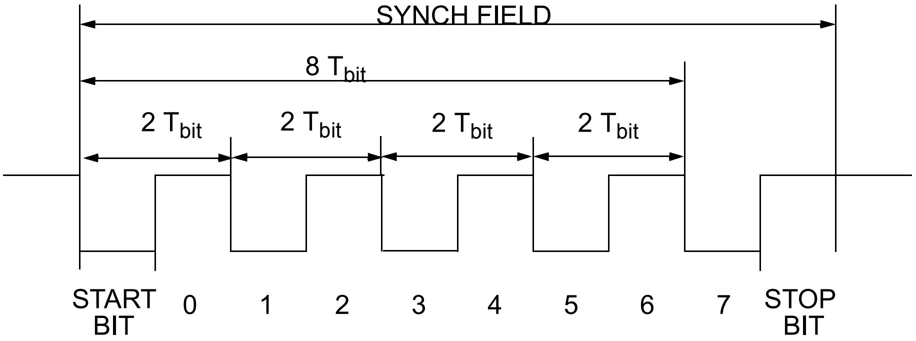


Figure 9.1: SYNCHRONIZATION FIELD

It is recommended to measure the time between the falling edges of both, the start bit and bit 7, and to divide the obtained value by 8. For the division by 8 it is recommended to shift the binary timer value by three positions towards LSB, and to use the first insignificant bit to round the result.

10 LINE DRIVER/RECEIVER

10.1 GENERAL CONFIGURATION

The bus line driver/receiver is an enhanced implementation of the ISO 9141 standard [2]. It consists of the bidirectional bus line LIN which is connected to the driver/receiver of every bus node, and is connected via a termination resistor and a diode to the positive battery node V_{BAT} (see **Figure 10.1**). The diode is mandatory to prevent an uncontrolled power-up of the ECU from the bus in case of a ’loss of battery’.

It is important to note that the LIN specification refers to the voltages at the external electrical connections of the electronic control unit (ECU), and not to ECU internal voltages. In particular the parasitic voltage drops of reverse polarity diodes have to be taken into account when designing a LIN transceiver circuit.

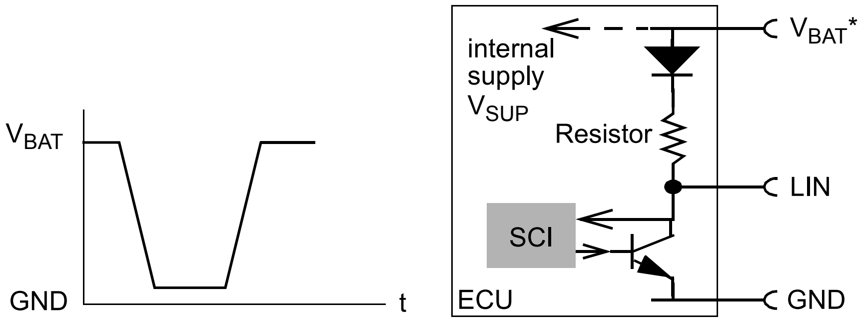


Figure 10.1: Concept of the Single-Wire Automotive Bus Interface
(* see Appendix A.6)

10.2 SIGNAL SPECIFICATION

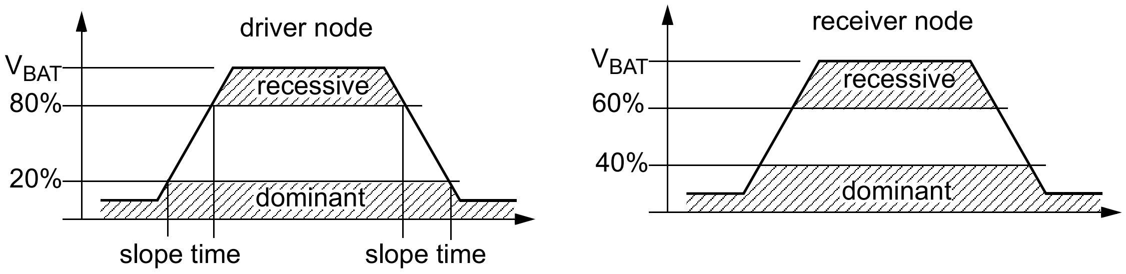


Figure 10.2: Voltage Levels on the Bus Line

The electrical DC parameters of the LIN Physical Layer and the termination resistors are listed in **Table 10.1** and **Table 10.2**, respectively. Note that in case of an integrated resistor/diode network no parasitic current paths must be formed between the bus line and the ECU-internal supply (V_{SUP}), for example by ESD elements.

parameter	**min.**	**typ.**	**max.**	**unit**	**comment**
V_{BAT} [a]	8		18	V	operating voltage range
$V_{BAT\_NON\_OP}$	-0.3		40	V	voltage range within which the device is not destroyed
I_{BUS} [b] @ V_{BUS}=1.2V	40		200	mA	dominant state (driver on)[c]
I_{BUS}	-1.1*V_{BAT}/R				dominant state (driver off) R := pull-up resistance as specified in **Table 10.2**
I_{BUS} @ V_{BUS} = V_{BAT} 8V<V_{BAT}<18V			20	µA	recessive state. Also applies if V_{BUS}>V_{BAT}.
I_{BUS} @ -12V<V_{BUS}<0V control unit disconnected from ground	-1		1	mA	loss of local ground must not affect communication in the residual network.
I_{BUS} @ -18V<V_{BUS}<-12V control unit disconnected from ground					Node has to sustain the current that can flow under this condition. Bus must be operational under this condition.
V_{BUSdom}	-8		0.4*V_{BAT}	V	receiver dominant state
V_{BUSrec}	0.6*V_{BAT}		18	V	receiver recessive state

Table 10.1: Electrical DC Parameters of the LIN Physical Layer

a. V_{BAT} denotes the supply voltage at the connector of the control unit and may be different from the internal supply V_{SUP} for electronic components (see Appendix A.6).

b. I_{BUS}: Current flowing into the node

c. A transceiver must be capable to sink at least 40mA. The maximum current flowing into the node must not exceed 200mA to avoid possible damage.

parameter	**min.**	**typ.**	**max.**	**unit**	**comment**
R_{master}	900	1000	1100	Ω	The serial diode is mandatory (**Figure 10.1**).
R_{slave}	20	30	47	KΩ	The serial diode is mandatory.

Table 10.2: Parameters of the Pull-Up Resistors

The electrical AC parameters of the LIN Physical Layer are listed in **Table 10.3**, with the timing parameters being defined in **Figure 10.3**.

parameter	min.	typ.	max.	unit	comment
| dV/dt | rising and falling edges (slew rate)	1	2	3	V/µs	The EMI behavior of the LIN bus depends on the signal slew rate, among other factors such as di/dt and d²V/dt². The value of the slew rate should be close to 2 V/µs to reduce emissions on the one hand and allow for speeds up to 20 kBit/sec on the other.
t_{trans_pd} propagation delay of transmitter			4	µs	see **Figure 10.3** $t_{trans\_pd}=\max(t_{trans\_pdr}, t_{trans\_pdf})$
t_{rec_pd} propagation delay of receiver			6	µs	see **Figure 10.3** $t_{rec\_pd} = \max(t_{rec\_pdr}, t_{rec\_pdf})$
t_{rec_sym} symmetry of receiver propagation delay rising edge w.r.t. falling edge	-2		2	µs	see **Figure 10.3** $t_{rec\_sym}=t_{rec\_pdf}-t_{rec\_pdr}$
t_{trans_sym} symmetry of transmitter propagation delay rising edge w.r.t falling edge	-2		2	µs	see **Figure 10.3** $t_{trans\_sym}=t_{trans\_pdf}-t_{trans\_pdr}$
t_{therm} Short circuit recovery time	1.5			ms	After detection of a short circuit the transmitter must be allowed to cool down again. Therefore the transmitter circuit must not be actuated within this time.

Table 10.3: Electrical AC Parameters of the LIN Physical Layer

Timing diagram:

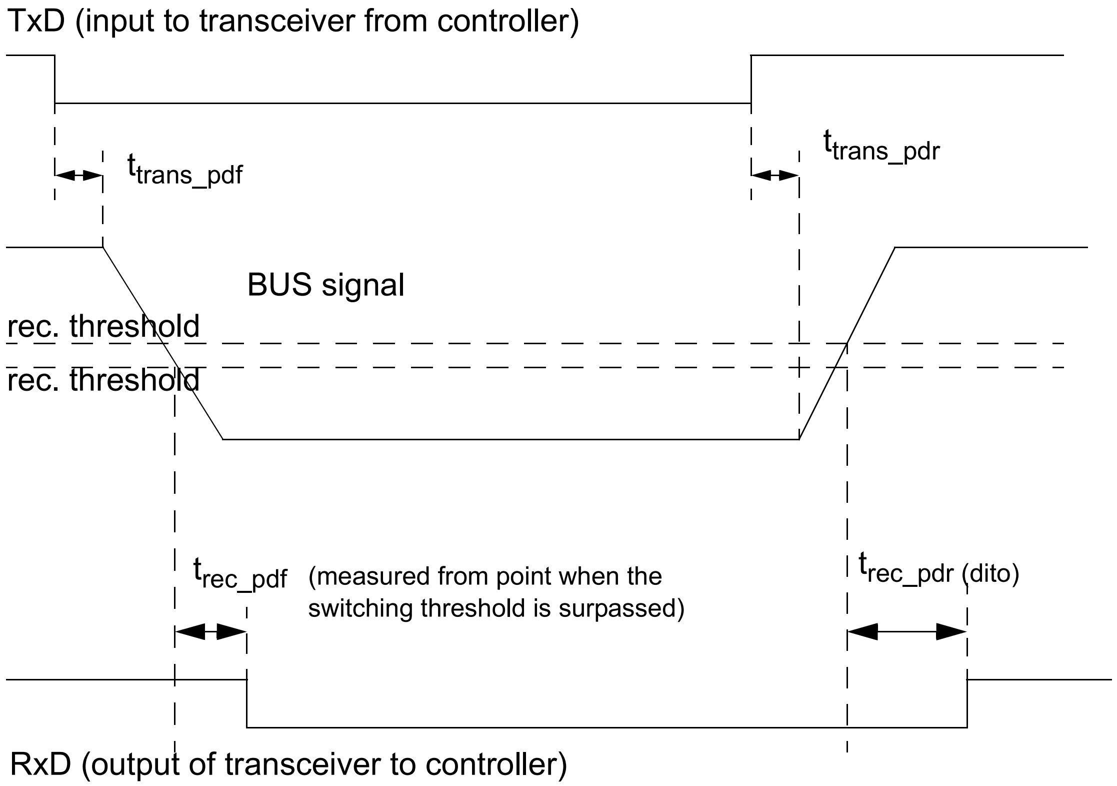


Figure 10.3: Definition of the BUS Timing

10.3 LINE CHARACTERISTICS

The maximum slew rate of rising and falling bus signals are in practice limited by the active slew rate control of typical bus transceivers. The minimum slew rate for the rising signal, however, can be given by the RC time constant. Therefore, the bus capacitance should be kept low in order not to keep the waveform asymmetry small. The capacitance of the master module can be chosen higher than in the slave modules, in order to provide a 'buffer' in case of network variants with various number of nodes. The total bus capacitance C_{BUS} can be calculated by **Equation 1** as

$$C_{BUS} = C_{MASTER} + n \cdot C_{SLAVE} + \overline{C}_{LINE} \cdot LEN_{BUS} \qquad \text{Equation 1}$$

under consideration of the parameters given in **Table 10.4**.

		typical	max	unit
total length of bus line	LEN_{BUS}		40	m
total capacitance of the bus including slave and master capacitances	C_{BUS}	4	10	nF
capacitance of master node	C_{MASTER}	220	2500	pF
capacitance of slave node	C_{SLAVE}	220	250	pF
line capacitance	$\overline{C}_{LINE}$	100	150	pF/m

Table 10.4: Line Characteristics and Parameters.

10.4 ESD/EMI COMPLIANCE

Semiconductor Physical Layer devices must comply with requirements for protection against human body discharge according to IEC 1000-4-2:1995. The minimum discharge voltage level is ± 2000V.

Note: The required ESD level for automotive applications can be up to ± 8000V at the connectors of the ECU.

11 REFERENCES

[1] J.W. Specks, A, Rajnák, “LIN - Protocol, Development Tools, and Software Interfaces for Local Interconnect Networks in Vehicles”, *9th Congress on Electronic Systems for Vehicles*, Baden-Baden, Germany, Oct. 5/6, 2000

[2] “Road vehicles - Diagnostic systems - Requirement for interchange of digital information”, *International Standard ISO9141*, 1st Edition, 1989

[3] Robert Bosch GmbH, "CAN Specification", *Version 2.0, Part B*, Stuttgart, 1991

A APPENDIX

A.1 EXAMPLES FOR MESSAGE SEQUENCES

A.1.1 Periodic Message Transfer

A regular message transfer on the bus might look such as

```
<MF 1> <IF-Space> <MF 2> <IF-Space> ... <IF-Space> <MF n> <IF-Space>
<MF 1> <IF-Space> <MF 2> <IF-Space> ... <IF-Space> <MF n> <IF-Space>
<MF 1> <IF-Space> <MF 2> <IF-Space> ... <IF-Space> <MF n> <IF-Space>
....
[MF = Message Frame; IF-Space = InterFrame Space]
```

This gives the opportunity to have a predictable worst case timing.

A.1.2 Bus Wake-Up Procedure

During the sleep mode, there is no bus activity. Any slave node is allowed to terminate the sleep mode by sending a WAKE-UP SIGNAL. In the regular case, the master node will than start the message transfer with a synch break:

```
[SLEEP MODE] [NODE-INTERNAL WAKE-UP] <WAKE-UP SIGNAL>
<MF 1> <IF-Space> <MF 2> <IF-Space> ... <IF-Space> <MF n> <IF-Space>
<MF 1> <IF-Space> <MF 2> <IF-Space> ... <IF-Space> <MF n> <IF-Space>
....
```

In case the master node does not respond, the WAKE-UP attempt is repeated two more times by the slave. The attempt for wake up is then suspended for a certain time before it can be resumed:

```
[SLEEP MODE] [NODE-INTERNAL WAKE-UP]
<WAKE-UP SIGNAL> <TIME-OUT AFTER BREAK>
<WAKE-UP SIGNAL> <TIME-OUT AFTER BREAK>
<WAKE-UP SIGNAL> <TIME-OUT AFTER THREE BREAKS>
[REPEAT BUS WAKE-UP PROCEDURE IF STILL PENDING]
```

A.2 TABLE OF VALID ID-FIELD VALUES

ID[0..5]		P0=	P1=	ID-Field	ID-Field		# of Data Bytes
Dec	Hex	ID0⊕ID1⊕ID2⊕ID4	$\overline{ID1 \oplus ID3 \oplus ID4 \oplus ID5}$	7 6 5 4 3 2 1 0	Dec	Hex	
0	0x00	0	1	1 0 0 0 0 0 0 0	128	0x80	2
1	0x01	1	1	1 1 0 0 0 0 0 1	193	0xC1	2
2	0x02	1	0	0 1 0 0 0 0 1 0	66	0x42	2
3	0x03	0	0	0 0 0 0 0 0 1 1	3	0x03	2
4	0x04	1	1	1 1 0 0 0 1 0 0	196	0xC4	2
5	0x05	0	1	1 0 0 0 0 1 0 1	133	0x85	2
6	0x06	0	0	0 0 0 0 0 1 1 0	6	0x06	2
7	0x07	1	0	0 1 0 0 0 1 1 1	71	0x47	2
8	0x08	0	0	0 0 0 0 1 0 0 0	8	0x08	2
9	0x09	1	0	0 1 0 0 1 0 0 1	73	0x49	2
10	0x0A	1	1	1 1 0 0 1 0 1 0	202	0xCA	2
11	0x0B	0	1	1 0 0 0 1 0 1 1	139	0x8B	2
12	0x0C	1	0	0 1 0 0 1 1 0 0	76	0x4C	2
13	0x0D	0	0	0 0 0 0 1 1 0 1	13	0x0D	2
14	0x0E	0	1	1 0 0 0 1 1 1 0	142	0x8E	2
15	0x0F	1	1	1 1 0 0 1 1 1 1	207	0xCF	2
16	0x10	1	0	0 1 0 1 0 0 0 0	80	0x50	2
17	0x11	0	0	0 0 0 1 0 0 0 1	17	0x11	2
18	0x12	0	1	1 0 0 1 0 0 1 0	146	0x92	2
19	0x13	1	1	1 1 0 1 0 0 1 1	211	0xD3	2
20	0x14	0	0	0 0 0 1 0 1 0 0	20	0x14	2
21	0x15	1	0	0 1 0 1 0 1 0 1	85	0x55	2
22	0x16	1	1	1 1 0 1 0 1 1 0	214	0xD6	2
23	0x17	0	1	1 0 0 1 0 1 1 1	151	0x97	2
24	0x18	1	1	1 1 0 1 1 0 0 0	216	0xD8	2
25	0x19	0	1	1 0 0 1 1 0 0 1	153	0x99	2
26	0x1A	0	0	0 0 0 1 1 0 1 0	26	0x1A	2
27	0x1B	1	0	0 1 0 1 1 0 1 1	91	0x5B	2
28	0x1C	0	1	1 0 0 1 1 1 0 0	156	0x9C	2
29	0x1D	1	1	1 1 0 1 1 1 0 1	221	0xDD	2
30	0x1E	1	0	0 1 0 1 1 1 1 0	94	0x5E	2
31	0x1F	0	0	0 0 0 1 1 1 1 1	31	0x1F	2
32	0x20	0	0	0 0 1 0 0 0 0 0	32	0x20	4
33	0x21	1	0	0 1 1 0 0 0 0 1	97	0x61	4
34	0x22	1	1	1 1 1 0 0 0 1 0	226	0xE2	4

Table A:2.1: Valid ID-Field Values

ID[0..5] Dec	Hex	P0= ID0⊕ID1⊕ID2⊕ID4	P1= $\overline{\text{ID1⊕ID3⊕ID4⊕ID5}}$	ID-Field 7 6 5 4 3 2 1 0	ID-Field Dec	Hex	# of Data Bytes
35	0x23	0	1	1 0 1 0 0 0 1 1	163	0xA3	4
36	0x24	1	0	0 1 1 0 0 1 0 0	100	0x64	4
37	0x25	0	0	0 0 1 0 0 1 0 1	37	0x25	4
38	0x26	0	1	1 0 1 0 0 1 1 0	166	0xA6	4
39	0x27	1	1	1 1 1 0 0 1 1 1	231	0xE7	4
40	0x28	0	1	1 0 1 0 1 0 0 0	168	0xA8	4
41	0x29	1	1	1 1 1 0 1 0 0 1	233	0xE9	4
42	0x2A	1	0	0 1 1 0 1 0 1 0	106	0x6A	4
43	0x2B	0	0	0 0 1 0 1 0 1 1	43	0x2B	4
44	0x2C	1	1	1 1 1 0 1 1 0 0	236	0xEC	4
45	0x2D	0	1	1 0 1 0 1 1 0 1	173	0xAD	4
46	0x2E	0	0	0 0 1 0 1 1 1 0	46	0x2E	4
47	0x2F	1	0	0 1 1 0 1 1 1 1	111	0x6F	4
48	0x30	1	1	1 1 1 1 0 0 0 0	240	0xF0	8
49	0x31	0	1	1 0 1 1 0 0 0 1	177	0xB1	8
50	0x32	0	0	0 0 1 1 0 0 1 0	50	0x32	8
51	0x33	1	0	0 1 1 1 0 0 1 1	115	0x73	8
52	0x34	0	1	1 0 1 1 0 1 0 0	180	0xB4	8
53	0x35	1	1	1 1 1 1 0 1 0 1	245	0xF5	8
54	0x36	1	0	0 1 1 1 0 1 1 0	118	0x76	8
55	0x37	0	0	0 0 1 1 0 1 1 1	55	0x37	8
56	0x38	1	0	0 1 1 1 1 0 0 0	120	0x78	8
57	0x39	0	0	0 0 1 1 1 0 0 1	57	0x39	8
58	0x3A	0	1	1 0 1 1 1 0 1 0	186	0xBA	8
59	0x3B	1	1	1 1 1 1 1 0 1 1	251	0xFB	8
60[a]	*0x3C*	*0*	*0*	*0 0 1 1 1 1 0 0*	*60*	*0x3C*	*8*
61[b]	*0x3D*	*1*	*0*	*0 1 1 1 1 1 0 1*	*125*	*0x7D*	*8*
62[c]	*0x3E*	*1*	*1*	*1 1 1 1 1 1 1 0*	*254*	*0xFE*	*8*
63[d]	*0x3F*	*0*	*1*	*1 0 1 1 1 1 1 1*	*191*	*0xBF*	*8*

Table A:2.1: (Continued)Valid ID-Field Values

a. Identifier 60 (0x3C) is reserved for the Master Request command frame (see Section 3.2).

b. Identifier 61 (0x3D) is reserved for the Slave Response command frame.

c. Identifier 62 (0x3E) is reserved for the user-defined extended frame (see Section 3.2).

d. Identifier 63 (0x3F) is reserved for a future LIN extended format.

A.3 EXAMPLE FOR CHECKSUM CALCULATION

Assumption:

Message frame with 4 data bytes.

`Data0 = 0x4A`
`Data1 = 0x55`
`Data2 = 0x93`
`Data3 = 0xE5`

	hex	**CY**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
0x4A	0x4A		0	1	0	0	1	0	1	0
+0x55 =	0x9F	0	1	0	0	1	1	1	1	1
(Add Carry)	0x9F		1	0	0	1	1	1	1	1
+0x93 =	0x132	1	0	0	1	1	0	0	1	0
Add Carry	0x33		0	0	1	1	0	0	1	1
+0xE5 =	0x118	1	0	0	0	1	1	0	0	0
Add Carry	0x19		0	0	0	1	1	0	0	1
Invert	0xE6		1	1	1	0	0	1	1	0
0x19+0xE6 =	0xFF		1	1	1	1	1	1	1	1

The resulting checksum is 0x19. The checkbyte is the inverted checksum 0xE6.

The receiving node can easily check the consistency of the data and the checkbyte by using the same addition mechanism. Checksum + checkbyte must result in 0xFF.

A.4 CAUSES FOR MESSAGE ERRORS

The following error mechanisms can cause the corruption of a message:

Local Disturbance of Ground Potential

The local ground potential of a receiver is below that of the sender so that a dominant bus voltage (logic level ’0’) is interpreted by the receiving node as recessive (logic level ’1’) or invalid. The input signal level is above the valid range for dominant signals. The cause of a ground shift can be for example high load currents through parasitic impedances of ground interconnections.

This disturbance is not detected by the bus voltage monitor of the sending node.

Local Disturbance of Supply Voltage

The local supply potential of a receiver is above that of the transmitter so that a recessive bus voltage (logic level ’1’) is interpreted by receiving node as dominant (logic level ’0’) or invalid. The input signal level is below the valid range for recessive signals. The cause of a local voltage rise can for example be a diode-capacitor voltage buffer for the internal electronics supply. In case of a voltage drop in the newtwork, the capacitor keeps the receiver-internal supply voltage temporarily above the sender-internal supply voltage.

This disturbance is not detected by the bus voltage monitor of the sending node.

Global Electric Disturbance of the Bus Signal

The voltage on the bus line is disturbed by for example electromagnetic interference such that the logical bus value is falsified.

This disturbance is detected by the bus voltage monitor of the sending node.

Unsynchronized Time Base

The time base of a slave node deviates significantly from that of the master time base so that incoming data bits are not sampled or outgoing data bits are not sent within the defined bit timing windows (see Section 9).

This disturbance is not detected by the bus voltage monitor of the sending node. A sending slave receives its own message correctly, while the master or any other slave receives an incorrect message that is sent with the 'wrong frequency'.

A.5 PROPOSALS FOR FAULT CONFINEMENT

Particular fault confinement is not part of hte LIN Protocol Specification. In the case fault confinement is implemented, teh following procedures are recommended:

A.5.1 Master Control Unit

- Master task sending: A Bit-Error in synchronization or identifier byte is detected when reading back the own transmission.

  The master control unit keeps track on any corrupted transmission by incrementing the MasterTransmitErrorCounter. This counter is increased by 8 each time the sent synchronization or identifier field is locally corrupted. It is decreased by 1 (not below 0) each time both fields are read back properly.

  If the counter is incremented beyond C_MASTER_TRANSMIT_ERROR-THRESHOLD it is assumed that there is a massive disturbance on the bus and an error handling procedure will take place on the application level.
- Slave task in the master control unit sending:
  A Bit-Error in a data or checksum field while reading back the own transmission.
- Slave task in the master control unit receiving: A Slave-Not-Responding-Error or a Checksum-Error is detected when expecting or reading a data from the bus.

  The master control unit keeps track on every corrupted transmission by incrementing the MasterReceiveErrorCounter [Number_of_Slave_Nodes] that is provided for every possible slave in the network. This counter is increased by 8 each time there are no valid data or checksum fields received. It is decreased by 1 (not below 0) each time the fields are received properly.

  If the counter is incremented beyond C_MASTER_RECEIVE_ERROR-THRESHOLD it is assumed that the addressed slave does not work properly and an error handling procedure will take place on the application level.

Error Variables	Recommended Default Value
C_MASTER_TRANSMIT_ERROR_THRESHOLD	64
C_MASTER_RECEIVE_ERROR-THRESHOLD	64

Table A:5.1: Error Variables for Fault Confinement

A.5.2 Slave Control Unit

- Slave task sending:
  A Bit-Error in a data or checksum field while reading back the own transmission.
- Slave task receiving:
  A Checksum-Error is detected while reading from the bus. The slave increments its error counter by eight and assumes that the other sending node is corrupted if it is only the messages generated by a particular node (this should be detectable by the master as well). If all messages seem to be corrupted an error in its own receiver circuitry is assumed. On a correct received messaged the error counter is decremented by one.

  Depending on the application, the response part of a message (data and checksum fields) can but needs not to be processed if this information is irrelevant for the application of this control unit. In such a case e.g. the checksum calculation can be omitted.

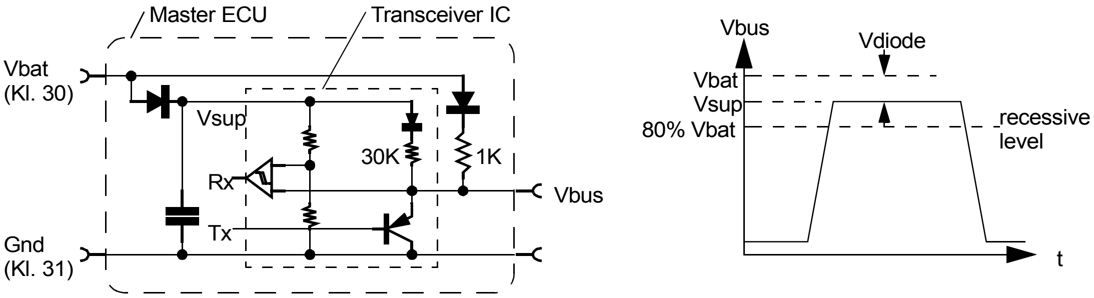
  If the slave does not see any bus traffic for a period specified in Section 6.1 (No-Bus-Activity) it is assumed, that the master is not alive. Depending on the error handling a wake-up sequence could be started or the slave changes to 'limp-home' mode.

  The slave does not see any valid sync-message but bus traffic for as described in Section 3.3 and in Section 6.1. The assumption is that the internal clock is far out of range. The slave should try to re-initialize, and if not possible go to limp-home mode. Since the slave does not reply to any message further error handling will be done by the master.

  The slave is not addressed for a time but receives valid sync-messages. The assumption is that the master does not request any service from the slave. So the slave should change to limp-home mode.

A.6 DEFINITION OF SUPPLY VOLTAGES FOR THE PHYSICAL INTERFACE

VBAT denotes the supply voltage at the connector of the control unit. Electronic components within the unit may see an internal supply VSUP being different from VBAT (see **Figure A.6.1**). This can be the result of protection filter elements and dynamic voltage changes on the bus. This has to be taken into consideration for the implementation of semiconductor products for LIN.



Vsup: internal supply for electronics

Figure A.6.1: Illustration of the Difference between External Supply Voltage VBAT and the Internal Supply Voltage VSUP

LIN

Configuration Language Specification

Revision 1.2

The following companies have provided advice for the contents of the Configuration Language Specification:
Audi AG, BMW AG, DaimlerChrysler AG, Motorola, Inc,
Volcano Communications Technologies AB, Volkswagen AG, Volvo Car Corporation

Responsible: István Horváth, Volcano Communications Technologies AB, Gruvgatan 35 A SE-421 30 V. Frölunda, SWEDEN
Ph: +46 31 451 116, Fax: +46 31 451 126, E-Mail: info@vct.se

Table of contents

1 Introduction

This document is part of the LIN specification.

1.1 What is the purpose of this document?

This document describes the syntax and semantics of the LIN description language recognised by a LIN Tool.

2 Revision history

Revision	Author	Date	Description
1.0	VCT-IHt	99-07-02	the first release of this specification
1.1	VCT-IHt	99-12-14	improvement in syntax, and one error corrected
1.11	VCT-IHt	00-02-11	removed previous chapter 8
1.2	VCT-IHt	00-08-28	event triggered frame definition was added
1.2	VCT-IHt	00-11-13	optional frame length definition was added
draft 2			diagnostic frame handling was added
			event triggered frame definition was changed

2.1 Differences between rev 1.0 and 1.1

- LIN protocol and language version number changed in grammar to be char_string.
- Error in description of group_offset corrected.
- Grammar for the encoding types improved.
- Example update according to the changes.

2.2 Differences between rev 1.1 and 1.11

- Emulation control file description removed
- Error in schedule table example corrected

2.3 Differences between rev 1.11 and 1.2

- The event triggered frame definition was added as option.
- Optional frame length definition was added
- Event triggered frame definition was changed
- Diagnostic address definition was added
- Diagnostic frame definition was added

3 References

Ref.	Document	Doc.nr	Rev./Date
[1]	LIN Protocol Specification		1.2
[2]	LIN API Recommended Practice		1.2

4 Terminology

4.1 Abbreviations

LIN	Local Interconnect Network
TBD	To be defined
Tool	LIN analyser/emulator

5 General scope

The language described in this document is used in order to create a “LIN description file”. The LIN description file describes a complete LIN network and also contains all information necessary to monitor the network. This information is sufficient to make a limited emulation (if the Tool supports that) controlled via the user’s interface of the Tool. (E.g. select emulated node(s), select schedule table.)

Neither syntax nor semantics are specified for the user’s interface of a LIN Tool, keeping the door open to Tool vendor specific implementations.

Furthermore the LIN description file can be one component used in order to write software for an electronic control unit which shall be part of the LIN network. An application programmer’s interface (API) has been defined (see reference [2]) as a recommended practice, in order to have a uniform way to access the LIN network from within different application programs. However, the functional behaviour of the application program is not address by the LIN description file.

6 Overview of Syntax

The syntax is described using a modified BNF (Bachus-Naur Format), as summarised below.

Symbol	Meaning
`::=`	A name on the left of the ::= is expressed using the syntax on its right.
`<>`	Used to mark objects specified later.
`|`	The vertical bar indicates choice. Either the left-hand side or the right hand side of the vertical bar shall appear.
`Bold`	The text in **bold** is reserved – either because it is a reserved word, or mandatory punctuation.
`[ ]`	The text between the square brackets shall appear once or several times.
`( )`	Used to group together some optional clauses.
`char_string`	Any character string enclosed in quotes “like this”.
`identifier`	An identifier. Typically used to name objects – identifiers shall follow the normal C rules for variable declaration.
`integer`	An integer. Integers can be in decimal (first digit is the range 1 to 9) or hexadecimal (prefixed with 0x).
`real_or_integer`	A real or integer number. A real number is always in decimal and has an embedded decimal point.

Within files using this syntax, comments are allowed anywhere. The comment syntax is the same as that for C++ where anything from // to the end of a line and anything enclosed in /* and */ delimiters shall be ignored.

7 LIN description file definition

```
<LIN_description_file> ::=
LIN_description_file ;
<LIN_protocol_version_def>
<LIN_language_version_def>
<LIN_speed_def>
<Node_def>
(<Diag_addr_def>)
<Signal_def>
(<Diag_signal_def>)
<Frame_def>
(<Event_triggered_frame_def>)
(<Diag_frame_def>)
<Schedule_table_def>
(<Signal_groups_def>)
(<Signal_encoding_type_def>)
(<Signal_representation_def>)
```

7.1 LIN protocol version number definition

```
<LIN_protocol_version_def> ::=
LIN_protocol_version = char_string ;
```

Shall be in the range of “0.01” to “99.99”.

7.2 LIN language version number definition

```
<LIN_language_version_def> ::=
LIN_language_version = char_string ;
```

Shall be in the range of “0.01” to “99.99”.

7.3 LIN speed definition

```
<LIN_speed_def> ::=
LIN_speed = real_or_integer kbps ;
```

Shall be in the range of 5.00 to 20.00 kilobit/second.

7.4 Node definition

```
<Node_def >::=
Nodes {
  Master:<node_name>,<time_base> ms ,<jitter> ms ;
  Slaves:<node_name>([,<node_name>]) ;
```

```
}
<node_name> ::= identifier
```

All `node_name` identifiers shall be unique within the `Nodes` sub-class.

The `node_name` identifier after the `Master` reserved word specifies the master node.

```
<time_base> ::= real_or_integer
```

The `time_base` value specifies the used time base in the master node to generate the maximum allowed frame transfer time. The time base shall be specified in milliseconds.

```
<jitter> ::= real_or_integer
```

The `jitter` value specifies the differences between the maximum and minimum delay from time base start point to the frame header sending start point (falling edge of BREAK signal). The jitter shall be specified in milliseconds. (For more information on `time_base` and `jitter` usage see the `Schedule_tables` sub-class definition.)

7.5 Node Diagnostic Address definition

```
<Diag_addr_def >::=
Diagnostic_addresses {
    [<node_name>:<diag_addr>;]
}
<node_name> ::= identifier
```

All `node_name` identifiers shall be equal to one of `node_name` identifier specified within the `Nodes` sub-class.

```
<diag_addr> ::= integer
```

The `diag_addr` specifies the diagnostic address for the identified node in the range of 1 to 255. (The diagnostic address 0 is reserved for future use.)

7.6 Signal definition

```
<Signal_def> ::=
Signals {
    [< signal_name >:< signal_size>,<init_value>,<published_by>
    [,<subscribed_by>];]
}
<signal_name> ::= identifier
```

All `signal_name` identifiers shall be unique within the `Signals` sub-class.

```
<signal_size> ::= integer
```

The `signal_size` specifies the size of the signal in 1 to 16 bits range.

```
<init_value> ::= integer
```

The `init_value` specifies the signal value that shall be used by all subscriber nodes until the frame containing the signal is received. The same initial signal value shall be sent from the publisher node (according the schedule table) until the application program has updated the signal.

```
<published_by> ::= identifier
<subscribed_by> ::= identifier
```

The `published_by` identifier and the `subscribed_by` identifier shall be equal to one of the `node_name` identifiers specified in the `Nodes` sub-class.

7.7 Frame definition

```
<Frame_def> ::=
Frames {
     [<frame_name>:<frame_id>,<published_by>(,<frame_size>) {
          [<signal_name>,<signal_offset>;]
     }]
}
<frame_name> ::= identifier
```

All `frame_name` identifiers shall be unique within the `Frames` sub-class.

```
<frame_id> ::= integer
```

The `frame_id` specifies the frame id number in range 0 to 63. The id shall be unique for all frames within the `Frames` sub-class.

The frame size is inherited from the frame id according to LIN protocol specification.

```
<published_by> ::= identifier
```

The `published_by` identifier shall be equal to one of the `node_name` identifiers specified in the `Nodes` sub-class.

```
<frame_size> ::= integer
```

The `frame_size` is an optional item, it specifies the size of the frame in range 1 to 8 bytes. If the `frame_size` specification not exists the size of the frame shall be derivated from the `frame_id` as is specified in the LIN protocol specification.

```
<signal_name> ::= identifier
```

The `signal_name` identifier shall be equal to one of the `signal_name` identifiers specified in the `Signals` sub-class.

All signals within one frame definition, shall be published by the same node as specified in the `published_by` identifier for that frame.

```
<signal_offset> ::= integer
```

The `signal_offset` value specifies the least-significant bit position of the signal in the frame. This value is in the range of 0 to (8 * frame_size – 1). The least significant bit of the signal is transmitted first.

Example:

The signal ‘S’ (size = 10 bits) is placed in a 4 bytes long frame with the offset 16. (The LSB of ‘S’ is at offset 16 and the MSB is at offset 25.)

Byte 0								Byte1								Byte2								Byte3							
																S	S	S	S	S	S	S	S	S	S						
0							7	8							15	16							23	24							31

transmitted first — transmitted last

The signal packing rules:

- Force (least significant bit) byte-alignment of all signals of size 8 bits and greater.
- Signals smaller than 8 bits are required to be contained within a single byte (thus, byte-boundaries can not be crossed by a “small” signal).

7.8 Event triggered frame definition

```
<Event_triggered_frame_def> ::=
Event_triggered_frames {
    [<event_trig_frm_name>:<frame_id>[,<frame_name>];]
}
<event_trig_frm_name> ::= identifier
```

All `event_trig_frm_name` identifiers shall be unique within the `Event_triggered_frames` sub-class and shall differ from all identifiers defined in the `Frames` sub-class.

```
<frame_id> ::= integer
```

The `frame_id` specifies the frame id number in range 0 to 63. The id shall be unique for all frames within the `Frames` and `Event_triggered_frames` sub-classes.

```
<frame_name> ::= identifier
```

All `frame_name` identifiers shall be equal with one of the identifiers specified within the `Frames` sub-class.

All frames specified by the `frame_name` identifier list shall be published by different nodes in the network.

NOTE: A frame specified by the `frame_name` in the `Frames` sub-class is additionally restricted in the layout, when it is mapped to an event triggered frame. An event triggered frame has the first byte reserved. This byte shall include the complete frame id (identifier and parity bits) of the frame specified within the `Frames` sub-class.

The slave task shall respond on the event triggered frame id (specified in the `Event_triggered_frames` sub-class) only if the frame's content was updated since previous transmission of the event triggered frame.

NOTE: The data contents of the event triggered frame and the frame specified in the `Frames` sub-class shall be identical when both frames are observed on the network. If more than one node is responding on an event triggered frame simultaneously, a bus-collision will occur. In this case the Master ECU is responsible to poll the single slave ECUs for all the mapped frames with frame id specified within the `Frames` sub-class.

7.9 Diagnostic frame definition

```
<Diag_frame_def> ::=
Diagnostic_frames {
    MasterReq : 60 {
          MasterReqB0,0;
          MasterReqB1,8;
          MasterReqB2,16;
          MasterReqB3,24;
          MasterReqB4,32;
          MasterReqB5,40;
          MasterReqB6,48;
          MasterReqB7,56;
    }
    SlaveResp : 61 {
          SlaveRespB0,0;
          SlaveRespB1,8;
          SlaveRespB2,16;
          SlaveRespB3,24;
          SlaveRespB4,32;
          SlaveRespB5,40;
          SlaveRespB6,48;
          SlaveRespB7,56;
    }
}
```

The `MasterReq` **and** `SlaveResp` reserved frame names are identifying the diagnostic frames.

The `MasterReq` frame has a fixed frame ID (60) and a fixed size (8 bytes) specified in the LIN protocol specification. The `MasterReq` frame can only be sent by the master node.

The `SlaveResp` frame has a fixed frame ID (61) and a fixed size (8 bytes) specified in the LIN protocol specification. The `SlaveResp` frame can only be sent by that slave node which was selected by the previous `MasterReq` frame. The selection of the slave node is based on the slave's diagnostic address specified in the `Diagnostic_addresses` sub-class.

The `MasterReqB0` **to** `MasterReqB7` reserved signal names are specifying the used signals within the `MasterReq` frame as 8 bits long integers.

The `SlaveRespB0` **to** `SlaveRespB7` reserved signal names are specifying the used signals within the `SlaveResp` frame as 8 bits long integers.

The `MasterReqB0` and `SlaveRespB0` shall be used as specified in the LIN protocol specification chapter 3.2.

The packing description of the predefined diagnostic signals (within the `MasterReq` **and** `SlaveResp` frame) is following the ordinary frame description syntax. (signal_name,signal_offset;)

7.10 Schedule table definition

```
<Schedule_table_def> ::=
Schedule_tables {
   [<schedule_table_name> {
        [<frame_name> delay <frame_time> ms ;]
    }]
}
<schedule_table_name> ::= identifier
```

All `schedule_table_name` identifier shall be unique within the `Schedule tables` sub-class.

```
<frame_name> ::= identifier
```

The `frame_name` identifier shall be equal to one of the `frame_name` identifiers specified in the `Frames` sub-class.

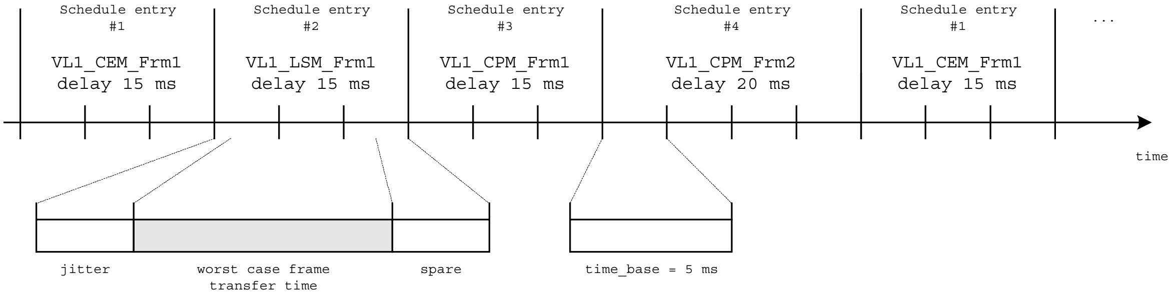
```
<frame_time> ::= real_or_integer
```

The `frame_time` specifies the time interval between two neighbouring frames. This time must be longer than the maximum allowed frame transfer time and it shall be exact multiple of the master node's time base value. The `frame_time` value shall be specified in milliseconds.

The schedule table selection shall be controlled by the Master application program. The switch between schedule tables must be done right after the frame time (for the currently transmitted frame) has elapsed.

Example:

```
Schedule_tables {
    VL1_ST1 {
           VL1_CEM_Frm1 delay 15 ms;
           VL1_LSM_Frm1 delay 15 ms;
           VL1_CPM_Frm1 delay 15 ms;
           VL1_CPM_Frm2 delay 20 ms;
    }
}
```



The delay specified for every schedule entry shall be longer than the `jitter` and the worst-case frame transfer time (see reference [1]).

Note! How to use and switch between different schedule tables is an application program issue, but the appropriate mechanisms are described in reference [2].

7.11 Signal groups definition

The signal groups sub-class is optional in the LIN file.

```
<Signal_groups_def> ::=
Signal_groups {
   [<signal_group_name>:<group_size> {
        [<signal_name>,<group_offset> ;]
    }]
}
<signal_group_name> ::= identifier
```

All `signal_group_name` identifier shall be unique within the `Signal_group` sub-class and shall be different from any `signal_name` identifier specified in the `Signals` sub-class.

```
<group_size> ::= integer
```

The `group_size` specifies the size of the signal in 1 to (8 * frame_size) bits range.

```
<signal_name> ::= identifier
```

The `signal_name` identifier shall be equal to one of the `signal_name` identifiers specified in the `Signals` sub-class.

```
<group_offset> ::= integer
```

The `group_offset` value specifies the least-significant bit position of the signal in the group. This value is in the range of `0` to (`group_size - 1`). The least significant bit is tramsmitted first. The not used bit positions in the group shall be filled with zeros.

One signal group shall always consist of signals residing in one single frame. The group definition gives a possibility to represent single signals larger than 16 bits in the Tool (still using a predefined specific signal encoding type).

7.12 Signal encoding type definition

The signal encoding type sub-class is optional in the LIN file.

```
<signal_encoding_type_def> ::=
Signal_encoding_types {
    [<signal_encoding_type_name> {
        [<logical_value> |
         <physical_range> |
         <bcd_range> |
         <ascii_range>]
    }]
}
<signal_encoding_type_name> ::= identifier
```

All `signal_encoding_type_name` identifier shall be unique within the `Signal encoding types` sub-class.

```
<logical_value> ::= logical_value,<signal_value>(,<text_info>) ;
<physical_range> ::= physical_value,<min_value>,<max_value>,<scale>,
<offset>(,<text_info>) ;
<bcd_value> ::= bcd_value ;
<ascii_value> ::= ascii_value ;
<signal_value> ::= integer
<min_value> ::= integer
<max_value> ::= integer
<scale> ::= real_or_integer
<offset> ::= real_or_integer
<text_info> ::= char_string
```

The `signal_value` the `min_value` and the `max_value` shall be in range of 0 to 65535. The `max_value` shall be greater than or equal to `min_value`. The signal encoding type information can be used by the Tool to replace a raw value with a logical/scaled physical value and/or with a predefined character string during monitoring. If the raw value is within the range defined by the min and max value, the physical value shall be calculated as:

physical_value = scale * raw_value + offset.

Examples:

```
Signal_encoding_types {
    1BitDig {
          logical_value,0,”off”;
          logical_value,1,”on”;
    }
    Temp {
          physical_value,0,250,0.5,-40,”degree”;
          physical_value,251,253,1,0,”undefined”;
          logical_value,254,”out of range”;
          logical_value,255,”error”;
    }
}
```

7.13 Signal representation definition

The signal representation sub-class is optional in the LIN file.

```
<Signal_representation_def> ::=
Signal_representation {
  [<signal_encoding_type_name>:<signal_or_group_name>
  ([,<signal_or_group_name>]);]
}
<signal_encoding_type_name> ::= identifier
```

The `signal_encoding_type_name` identifier shall be equal to one of the `signal_encoding_type_name` identifier specified in the `Signal_encoding_types` sub-class.

```
<signal_or_group_name> ::= <signal_name> | <signal_group_name>
<signal_name> ::= identifier
<signal_group_name> ::= identifier
```

The `signal_name` identifier shall be equal to one of the `signal_name` identifiers specified in the `Signals` sub-class.

The `signal_group_name` identifier shall be equal to one of the `signal_group_name` identifiers specified in the `Signal_groups` sub-class.

8 Examples

8.1 LIN description file

```
// This is a LIN description example file
// Issued by Istvan Horvath
LIN_description_file ;
LIN_protocol_version = “1.0”;
LIN_language_version = “1.1”;
LIN_speed = 19.2 kbps;

Nodes {
    Master:CEM,5 ms, 0.1 ms;
    Slaves:LSM,CPM;
}

Signals {
    RearFogLampInd:1,0,CEM,LSM;
    PositionLampInd:1,0,CEM,LSM;
    FrontFogLampInd:1,0,CEM,LSM;
    IgnitionKeyPos:3,0,CEM,LSM,CPM;
    LSMFuncIllum:4,0,CEM,LSM;
    LSMSymbolIllum:4,0,CEM;
    StartHeater:3,0,CEM;
    CPMReqB0:8,0,CEM;
    CPMReqB1:8,0,CEM;
    CPMReqB2:8,0,CEM;
    CPMReqB3:8,0,CEM;
    CPMReqB4:8,0,CEM;
    CPMReqB5:8,0,CEM;
    CPMReqB6:8,0,CEM;
    CPMReqB7:8,0,CEM;
    ReostatPos:4,0,LSM;
    HeadLampBeamLev:4,0,LSM;
    FrontFogLampSw:1,0,LSM;
    RearFogLampSw:1,0,LSM;
    MLSOff:1,0,LSM;
    MLSHeadLight:1,0,LSM;
    MLSPosLight:1,0,LSM;
    HBLSortHigh:1,0,LSM;
    HBLShortLow:1,0,LSM;
```

```
        ReoShortHigh:1,0,LSM;
        ReoShortLow:1,0,LSM;
        LSMHWPartNoB0:8,0,LSM;
        LSMHWPartNoB1:8,0,LSM;
        LSMHWPartNoB2:8,0,LSM;
        LSMHWPartNoB3:8,0,LSM;
        LSMSWPartNo:8,0,LSM;
        CPMOutputs:10,0,CPM;
        HeaterStatus:4,0,CPM;
        CPMGlowPlug:7,0,CPM;
        CPMFanPWM:8,0,CPM;
        WaterTempLow:8,0,CPM;
        WaterTempHigh:8,0,CPM;
        CPMFuelPump:7,0,CPM;
        CPMRunTime:13,0,CPM;
        FanIdealSpeed:8,0,CPM;
        FanMeasSpeed:8,0,CPM;
        CPMRespB0:1,0,CPM;
        CPMRespB1:1,0,CPM;
        CPMRespB2:1,0,CPM;
        CPMRespB3:1,0,CPM;
        CPMRespB4:1,0,CPM;
        CPMRespB5:1,0,CPM;
        CPMRespB6:1,0,CPM;
        CPMRespB7:1,0,CPM;
    }

    Frames {
        VL1_CEM_Frm1:32,CEM {
              RearFogLampInd,0;
              PositionLampInd,1;
              FrontFogLampInd,2;
              IgnitionKeyPos,3;
              LSMFuncIllum,8;
              LSMSymbolIllum,12;
              StartHeater,16;
        }
        VL1_CEM_Frm2:48,CEM {
              CPMReqB0,0;
              CPMReqB1,8;
              CPMReqB2,16;
              CPMReqB3,24;
              CPMReqB4,32;
              CPMReqB5,40;
              CPMReqB6,48;
```

```
                CPMReqB7,56;
        }
        VL1_LSM_Frm1:33,LSM {
                ReostatPos,0;
                HeadLampBeamLev,4;
                FrontFogLampSw,8;
                RearFogLampSw,9;
                MLSOff,10;
                MLSHeadLight,11;
                MLSPosLight,12;
                HBLSortHigh,16;
                HBLShortLow,17;
                ReoShortHigh,18;
                ReoShortLow,19;
        }
        VL1_LSM_Frm2:49,LSM {
                LSMHWPartNoB0,0;
                LSMHWPartNoB1,8;
                LSMHWPartNoB2,16;
                LSMHWPartNoB3,32;
                LSMSWPartNo,40;
        }
        VL1_CPM_Frm1:50,CPM {
                CPMOutputs,0;
                HeaterStatus,10;
                CPMGlowPlug,16;
                CPMFanPWM,24;
                WaterTempLow,32;
                WaterTempHigh,40;
                CPMFuelPump,56;
        }
        VL1_CPM_Frm2:34,CPM {
                CPMRunTime,0;
                FanIdealSpeed,16;
                FanMeasSpeed,24;
        }
        VL1_CPM_Frm3:51,CPM {
                CPMRespB0,0;
                CPMRespB1,8;
                CPMRespB2,16;
                CPMRespB3,24;
                CPMRespB4,32;
                CPMRespB5,40;
                CPMRespB6,48;
                CPMRespB7,56;
```

```
        }
    }

    Schedule_tables {
        VL1_ST1 {
              VL1_CEM_Frm1 delay 15 ms;
              VL1_LSM_Frm1 delay 15 ms;
              VL1_CPM_Frm1 delay 20 ms;
              VL1_CPM_Frm2 delay 20 ms;
        }
        VL1_ST2 {
              VL1_CEM_Frm1 delay 15 ms;
              VL1_CEM_Frm2 delay 20 ms;
              VL1_LSM_Frm1 delay 15 ms;
              VL1_LSM_Frm2 delay 20 ms;
              VL1_CEM_Frm1 delay 15 ms;
              VL1_CPM_Frm1 delay 20 ms;
              VL1_CPM_Frm2 delay 20 ms;
              VL1_LSM_Frm1 delay 15 ms;
              VL1_CPM_Frm3 delay 20 ms;
        }
    }
    Signal_groups {
        CPMReq:64 {
              CPMReqB0,0;
              CPMReqB1,8;
              CPMReqB2,16;
              CPMReqB3,24;
              CPMReqB4,32;
              CPMReqB5,40;
              CPMReqB6,48;
              CPMReqB7,56;
        }
    }

    Signal_encoding_types {
        1BitDig {
              logical_value,0,”off”;
              logical_value,1,”on”;
        }
        2BitDig {
              logical_value,0,”off”;
              logical_value,1,”on”;
              logical_value,2,”error”;
              logical_value,3,”void”;
```

```
        }
        Temp {
              physical_value,0,250,0.5,-40,"degree";
              physical_value,251,253,1,0,"undefined";
              logical_value,254,"out of range";
              logical_value,255,"error";
        }
        Speed {
              physical_value,0,65500,0.008,250,"km/h";
              physical_value,65501,65533,1,0,"undefined";
              logical_value,65534,"error";
              logical_value,65535,"void";
        }
    }

    Signal_representations {
        1BitDig:RearFogLampInd,PositionLampInd,FrontFogLampInd;
        Temp:WaterTempLow,WaterTempHigh;
        Speed:FanIdealSpeed,FanMeasSpeed;
    }
```

LIN API

Recommended Practice

Revision 1.2

The following companies have provided advice for the contents of the LIN API Recommended Practice:
Audi AG, BMW AG, DaimlerChrysler AG, Motorola, Inc,
Volcano Communications Technologies AB, Volkswagen AG, Volvo Car Corporation

Table of contents

7 EXAMPLES **20**

1 Introduction

This document defines a recommended practice for implementing an application programmer’s interface (API) to a LIN SW module, and serves as a complement to the LIN standard specification.

1.1 What is the purpose of this document?

The purpose of this document is to define a suitable interface for a LIN SW module to the application SW.

2 Revision history

Rev.	Author	Date	Description
1.0	VCT-CBn	00-02-10	The first release of this recommended practice
1.1	VCT-CBn	00-08-28	Updated in accordance with change request posted 00-07-03. Change in return value policy on `l_sch_tick` function and added input parameter to the `l_sch_set` function.
1.2	VCT-CBn	00-11-13	Updated in accordance with change request posted 00-09-26. Added return values to the `l_ifc_connect` and `l_ifc_disconnect` functions. Additional description text to the `l_ifc_init` function. Corrected error in type of the return value for the dynamic `l_ifc_ioctl` function.

3 References

Ref.	Document	Doc.nr	Rev./Date
[1]	LIN Configuration language specification		1.11

4 Terminology

4.1 Abbreviations

API	Application Program Interface
ECU	Electronic Control Unit (with µ-Controller/µ-Processor)
LCFG	LIN Configuration tool (PC-program)
LIN	Local Interconnect Network
TBD	To be defined

5 General

This chapter describes a possible solution, how to integrate an application program together with the LIN API.

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. Instead the user will be provided an API which is focused on the signals transported on the LIN network. A PC-tool is introduced, called the “LCFG” which will take care of the step from network configuration to ready made program code. This will provide the user with configuration flexibility.

The inputs of the LCFG will be one or many LIN network configuration files (described in reference [1]) and one local description file (separately described). The network configuration files will contain the total definition of a specific LIN network. The local file will describe node specific entities (e.g. flags connected to signals/frames and HW-specifications).

LCFG will generate ANSI C-files and H-files, which should be compiled together with the application SW. Furthermore the user, will include a provided LIN library into the application program. The workflow when using the LIN-library and API to create an application using LIN as communication interface is shown in Figure 1.

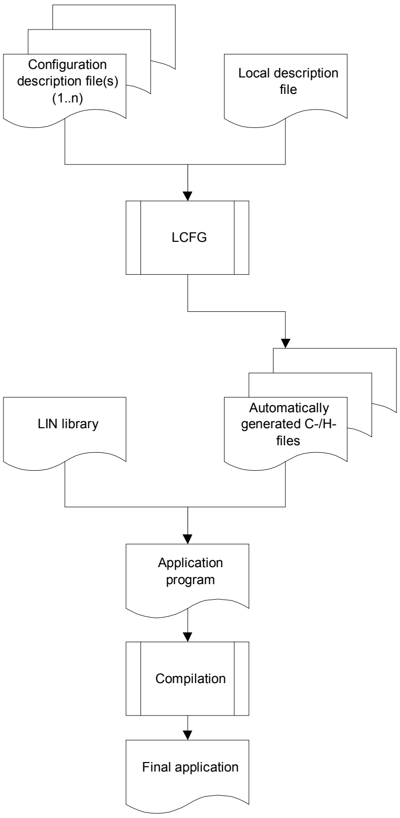


Figure 1: LIN configuration workflow

6 API specification

The LIN API has a set of functions all based on the idea to give the API a separate name-space, in order to minimise the risk of conflicts with existing SW. All functions and types will have the prefix “`l_`“ (lower-case “`L`” followed by an “underscore”).

The LIN SW will define the following types:

- `l_bool`
- `l_ioctl_op`
- `l_irqmask`
- `l_u8`
- `l_u16`

In order to gain efficiency, the majority of the functions will be so-called static functions (C pre-processor macro #define’s, automatically generated by the LIN configuration tool).

6.1 Initialisation

6.1.1 l_sys_init

Prototype

```
l_bool l_sys_init(void);
```

Description

`l_sys_init` performs the initialisation of the LIN SW.

Returns

zero if the initialisation succeeded and

non-zero if the initialisation failed

Notes

The call to the `l_sys_init` is the first call a user must use in the LIN SW before using any other API functions.

6.2 Signal calls

6.2.1 Signal types

The signals will be of three different types:

- `l_bool`: for one bit signals (zero if false, non-zero otherwise)
- `l_u8`: for signals of the size 1 – 8 bits
- `l_u16`: for signals of the size 9 – 16 bits

6.2.2 Read calls

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 implementation

```
l_bool l_bool_rd_sss(void);
l_u8 l_u8_rd_sss(void);
l_u16 l_u16_rd_sss(void);
```

Where `sss` is the name of the signal (e.g. `l_u8_rd_EngineSpeed()`).

Description

Reads and returns the current value of the signal specified by the name `sss`.

Notes

none

6.2.3 Write calls

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 implementation

```
void l_bool_wr_sss(l_bool v);
void l_u8_wr_sss(l_u8 v);
void l_u16_wr_sss(l_u16 v);
```

Where `sss` is the name of the signal (e.g. `l_u8_wr_EngineSpeed(v)`).

Description

Sets the current value of the signal specified by the name `sss` to the value `v`.

Notes

none

6.3 Flag calls

Flags are ECU local objects that are used to synchronise the application program with the LIN SW. The flags will be automatically set by the LIN SW, and can only be tested/cleared by the application program.

6.3.1 l_flg_tst

Dynamic prototype

```
l_bool l_flg_tst(l_flag_handle fff);
```

Static implementation

```
l_bool l_flg_tst_fff(void);
```

Where `fff` is the name of the flag (e.g. `l_flg_tst_RxEngineSpeed()`).

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).

Notes

none

6.3.2 l_flg_clr

Dynamic prototype

```
void l_flg_clr(l_flag_handle fff);
```

Static implementation

```
void l_flg_clr_fff(void);
```

Where `fff` is the name of the signal (e.g. `l_flg_clr_RxEngineSpeed()`).

Description

Sets the current value of the flag specified by the name `fff` to zero.

Notes

none

6.4 Processing calls

6.4.1 l_sch_tick

Dynamic prototype

```
l_u8 l_sch_tick(l_ifc_handle iii);
```

Static implementation

```
l_u8 l_sch_tick_iii(void);
```

Where `iii` is the name of the interface (e.g. `l_sch_tick_MyLinIfc()`).

Description

The `l_sch_tick` function follows a schedule. 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 last schedule set up by the last call to `l_sch_set`.

The `l_sch_tick` must be called individually for each interface within the ECU, with the rate specified in the network configuration file.

Returns

"non-zero" if the next call of `l_sch_tick` will start the transmission of the frame in the next schedule table entry. The return value will in this case be the next schedule table entry's number (counted from the beginning of the schedule table) in the schedule table. The return value will be in range 1 to 'N' if the schedule table has 'N' entries.

"zero" otherwise

Notes

`l_sch_tick` may only be used in the Master node.

The call to `l_sch_tick` will not only start the transition of the next frame due, it will also update the signal values for those signals received since the previous call to `l_sch_tick` (i.e. in the last frame on this interface).

(See also note on `l_sch_set` for use of return value.)

6.4.2 l_sch_set

Dynamic prototype

```
void l_sch_set(l_ifc_handle iii, l_schedule_handle sch, l_u8 ent);
```

Static implementation

```
void l_sch_set_iii(l_schedule_handle sch, l_u8 ent);
```

Where `iii` is the name of the interface (e.g. `l_sch_set_MyLinIfc(MySchedule1,0)`).

Description

Sets up the next schedule `sch` to be followed by the `l_sch_tick` function for a certain interface `iii`. The new schedule will be activated as soon as the current schedule reaches its next schedule entry point.

The input parameter `ent` defines the starting entry point in the new schedule table. The value of `ent` should be in the range 0 to ‘N’ if the schedule table has ‘N’ entries, and if `ent` is 0 or 1 the new schedule table will be started from the beginning.

Notes

`l_sch_set` may only be used in the Master node.

The `ent` input value in combination with the `l_sch_tick` return value can be used to e.g; 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.

6.5 Interface calls

6.5.1 l_ifc_init

Dynamic prototype

```
void l_ifc_init(l_ifc_handle iii);
```

Static implementation

```
void l_ifc_init_iii(void);
```

Where `iii` is the name of the interface (e.g. `l_ifc_init_MyLinIfc()`).

Description

`l_ifc_init` initialises the controller specified by the name `iii` (i.e. sets up internals such as the baud rate). The default schedule set by the `l_ifc_init` call will be a so-called `NULL_Schedule` where no frames will be sent and received.

Notes

The interfaces are all listed by their names in the local description file.

The call to the `l_ifc_init()` function is the first call a user must perform, before using any other interface-related LIN API functions (e.g. the `l_ifc_connect()` or `l_ifc_rx()`).

6.5.2 l_ifc_connect

Dynamic prototype

```
l_bool l_ifc_connect(l_ifc_handle iii);
```

Static implementation

```
l_bool l_ifc_connect_iii(void);
```

Where `iii` is the name of the interface (e.g. `l_ifc_connect_MyLinIfc()`).

Description

The call to the `l_ifc_connect` will connect the interface `iii` to the LIN network and enable the transmission of headers and data to the bus.

Returns

zero if the “connect operation” was successful and

non-zero if the “connect operation” failed

Notes

none

6.5.3 l_ifc_disconnect

Dynamic prototype

```
l_bool l_ifc_disconnect(l_ifc_handle iii);
```

Static implementation

```
l_bool l_ifc_disconnect_iii(void);
```

Where `iii` is the name of the interface (e.g. `l_ifc_disconnect_MyLinIfc()`).

Description

The call to the `l_ifc_disconnect` will disconnect the interface `iii` from the LIN network and thus disable the transmission of headers and data to the bus.

Returns

zero if the “disconnect operation” was successful and

non-zero if the “disconnect operation” failed

Notes

none

6.5.4 l_ifc_ioctl

Dynamic prototype

```
l_u16 l_ifc_ioctl(l_ifc_handle iii, l_ioctl_op op, void *pv);
```

Static implementation

```
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)`).

Description

This function controls protocol and interface specific parameters. The `iii` is the name of the interface to which the operation defined in `op` should be applied. The pointer `pv` points to an optional parameter block.

Exactly which operations that are supported, depends on the interface type and the programmer must therefore refer to the documentation for the specific interface in the target-binding document. This document will specify what all operations do, and the value returned.

Notes

The interpretation of the parameter block depends upon the operation chosen. Some operations do not need this block. In such cases the pointer `pv` can be set to `NULL`. In the cases where the parameter block is relevant its format depends upon the interface and, therefore, the interface specification the target-binding document must be consulted.

6.5.5 l_ifc_rx

Dynamic prototype

```
void l_ifc_rx(l_ifc_handle iii);
```

Static implementation

```
void l_ifc_rx_iii(void);
```

Where `iii` is the name of the interface (e.g. `l_ifc_rx_MyLinIfc()`).

Description

Called when the interface `iii` has received one character of data.

E.g. called from a user-defined interrupt handler raised by a UART when it receives one character of data. In this case the function will perform necessary operations on the UART control registers.

Notes

The application program is responsible for binding the interrupt and for setting the correct interface handle (if interrupt is used).

6.5.6 l_ifc_tx

Dynamic prototype

```
void l_ifc_tx(l_ifc_handle iii);
```

Static implementation

```
void l_ifc_tx_iii(void);
```

Where `iii` is the name of the interface (e.g. `l_ifc_tx_MyLinIfc()`).

Description

Called when the interface `iii` has transmitted one character of data.

E.g. called from a user-defined interrupt handler raised by a UART when it has transmitted one character of data. In this case the function will perform necessary operations on the UART control registers.

Notes

The application program is responsible for binding the interrupt and for setting the correct interface handle (if interrupt is used).

This function might even be empty in certain implementations, where the transmission is coupled to the `l_ifc_rx` function call. This is described for the user in the target-binding document.

6.5.7 l_ifc_aux

Dynamic prototype

```
void l_ifc_aux(l_ifc_handle iii);
```

Static implementation

```
void l_ifc_aux_iii(void);
```

Where `iii` is the name of the interface (e.g. `l_ifc_aux_MyLinIfc()`).

Description

This function may be used in the slave nodes to synchronise to the BREAK and SYNC characters sent by the master on the interface specified by `iii`.

E.g. called from a user-defined interrupt handler raised upon a flank-detection on a HW-pin connected to the interface `iii`.

Notes

`l_ifc_aux` may only be used in the Slave node.

This function is strongly HW-connected and the exact implementation and usage is described for the user in the target-binding document.

This function might even be empty in cases where the BREAK/SYNC detection is implemented in the `l_ifc_rx` function.

6.6 User provided call-outs

The user must provide a pair of functions, which will be called from within the LIN SW in order to disable all controller interrupts before certain internal operations, and to restore the previous state after such operations. (These functions are used e.g. in the l_sch_tick function.)

6.6.1 l_sys_irq_disable

Dynamic prototype

```
l_irqmask l_sys_irq_disable(void);
```

Description

The user implementation of this function must achieve a state in which no controller interrupts can occur.

Notes

none

6.6.2 l_sys_irq_restore

Dynamic prototype

```
void l_sys_irq_restore(l_irqmask previous);
```

Description

The user implementation of this function must restore the state identified by `previous`.

Notes

none

7 Examples

In the following chapters a very simple example is given in order to show how the API can be used. The C application code is shown as well as the LIN description file.

7.1 LIN API usage

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

/*         File: hello.c                                                    */
/*       Author: Christian Bondesson                                        */
/*  Description: Example code for using the LIN API in a LIN master ECU     */
/*               NOTE! This is using the static API!!!                      */
/*
**  $Header$
*/
/*  Date:   Author:     Description:                                        */
/*  -----   -------     ------------                                        */
/*  990830  VCT-CBn     * new created                                       */
/*  000828  VCT-CBn     * adopted to API version 1.1 (the l_sch_tick and    */
/*                      l_sch_set functions updated)                        */
/*  001113  VCT-CBn     * adopted to API version 1.2 (the l_ifc_connect     */
/*                      function updated with return value)                 */

/* include files: */
/* -------------- */
/* #include file */
#include "lin.h"

/* ************************************************************************ */
/*   PROCEDURE : l_sys_irq_restore                                          */
/* DESCRIPTION : Restores the interrupt mask to the one before the call to  */
/*               l_sys_irq_disable was made                                 */
/*          IN : previous - the old interrupt level                         */
/* ************************************************************************ */
void l_sys_irq_restore(l_imask previous)
{
    /* some controller specific things...                                   */

} /* end l_sys_irq_restore */

/* ************************************************************************ */
/*   PROCEDURE : l_sys_irq_disable                                          */
/* DESCRIPTION : Disable all interrupts of the controller and returns the   */
/*               interrupt level to be able to restore it later             */
/* ************************************************************************ */
l_imask l_sys_irq_disable(void)
{
    /* some controller specific things...                                   */

} /* end l_sys_irq_disable */

/* ************************************************************************ */
/*   INTERRUPT : lin_char_rx_handler                                        */
/* DESCRIPTION : LIN recieve character interrupt handler for the interface  */
/*               named LIN_ifc                                              */
/* ************************************************************************ */
void INTERRUPT lin_char_rx_handler(void)
{
                                    /* just call the LIN API provided func- */
                                    /* tion to do the actual work           */
    l_ifc_rx_MyLinIfc();

} /* end lin_char_rx_handler */
```

```
/* ************************************************************************ */
/*   INTERRUPT : lin_char_tx_handler                                        */
/* DESCRIPTION : LIN transmit character interrupt handler for the interface */
/*               named LIN_ifc                                              */
/* ************************************************************************ */
void INTERRUPT lin_char_tx_handler(void)
{
                                    /* just call the LIN API provided func- */
                                    /* tion to do the actual work           */
    l_ifc_tx_MyLinIfc();

} /* end lin_char_tx_handler */

/* ************************************************************************ */
/*   PROCEDURE : main                                                       */
/* DESCRIPTION : Main program... initialisation part                        */
/* ************************************************************************ */
void main(void)
{
                                    /* initialise the LIN interface         */
    if (l_sys_init())
    {
                                    /* the init of the LIN software failed  */
    }
    else
    {
        l_ifc_init_MyLinIfc();      /* initialise the interface             */
        if (l_ifc_connect_MyLinIfc())
        {
                                    /* connection of the LIN interface fai- */
                                    /* led                                  */
        }
        else
        {
                                    /* connected, now ready to send/receive */
                                    /* set the normal schedule to run from  */
                                    /* beginning for this specific inter-   */
                                    /* face                                 */
            l_sch_set_MyLinIfc(MySchedule1, 0);
        }
    }

    start_main_application();       /* ready with init, start actaul appl.  */

} /* end main */

/* 10ms based on the minimum LIN tick time, in LIN description file...      */
void main_application_10ms(void)
{
    /* do some application specific stuff...                                */
                                    /* just a small example of signal rea-  */
                                    /* ding and writing                     */
    if (l_flg_tst_RxInternalLightsSwitch())
    {
        l_u8_wr_InternalLightsRequest(l_u8_rd_InternalLightsSwitch());
        l_flg_clr_RxInternalLightsSwitch();
    }
                                    /* in-/output of signals, do not care   */
                                    /* about the return value, as we will   */
                                    /* never switch schedule anyway...      */
    (void)l_sch_tick_MyLinIfc();

} /* end main_application_10ms */
```

7.2 LIN description file

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

/*         File: hello.ldf                                                  */
/*       Author: Christian Bondesson                                        */
/*  Description: The LIN description file for the example program           */
/*
**  $Header$
*/
/*  Date:   Author:     Description:                                        */
/*  -----   -------     ------------                                        */
/*  990830  VCT-CBn     * new created                                       */

LIN_description_file ;
LIN_protocol_version = “1.0”;
LIN_language_version = “1.1”;
LIN_speed = 19.2 kbps;

Nodes {
    Master: CEM, 5 ms, 0.1 ms;
    Slaves: LSM;
}

Signals {
    InternalLightsRequest: 2, 0, CEM, LSM;
    InternalLightsSwitch: 2, 0, LSM, CEM;
}

Frames {
    VL1_CEM_Frm1: 1, CEM {
        InternalLightsRequest, 0;
    }
    VL1_LSM_Frm1: 2, LSM {
        InternalLightsSwitch, 0;
    }
}

Schedule_tables {
    MySchedule1 {
        VL1_CEM_Frm1 delay 15 ms;
        VL1_LSM_Frm1 delay 15 ms;
    }
}

Signal_encoding_types {
    2BitDig {
        logical_value, 0, “off”;
        logical_value, 1, “on”;
        logical_value, 2, “error”;
        logical_value, 3, “void”;
    }
}

Signal_representations {
    2BitDig: InternalLightsRequest, InternalLightsSwitch;
}
```